Determining metamodel appropriateness for describing a domain

Razenberg, F.R.M.

Award date: 2014

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Determining metamodel appropriateness for describing a domain

Master Thesis

Frank Razenberg

Supervisors:
dr.ir. R.R.H. Schiffelers
Y. Dajsuren PDEng

Eindhoven, February 2014
Abstract

At ASML, the CARM 2G framework is used to design control systems in the domain of their TWINSCAN lithography platform. In the automotive industry, EAST-ADL is one of the most researched architecture description languages for describing automotive embedded systems. Although CARM 2G and EAST-ADL are targeted at different domains, they both share concepts usable to model features in either domain. These range from hierarchical modeling levels describing different levels of refinement, to actual electrical components used within such layers.

A different application of EAST-ADL is found in ASML’s Hardware Software Interface. This is a large specification of available software interfaces on a hardware platform, which ASML modeled using the EAST-ADL based tool PREEvision.

ASML expressed the desire to learn whether EAST-ADL could be used for modeling the entire TWINSCAN domain, and to evaluate how suited EAST-ADL is for modeling the HSI in. We performed a feasibility study to enact whether the EAST-ADL metamodel sufficiently accommodates CARM 2G by re-implementing models of case studies in the EAST-ADL implementation PREEvision. This shows us how the metamodels of both architectures relate, and allowed us to extract a strategy to re-implement arbitrary CARM 2G models in PREEvision. Additionally, we studied the PREEvision implementation of the HSI and identified the shortcomings in this approach. We attempted to alleviate these by employing a custom metamodel specifically targeted at the HSI. We ensured compatibility with PREEvision models by means of a two-step model transformation.

In order to quantify the usability and effectiveness of the EAST-ADL metamodel for the CARM 2G and HSI domains, we introduced a novel metric which is able to quantify how well a metamodel fits an arbitrary domain. This confirmed our hypothesis that EAST-ADL does not properly accommodate the CARM 2G domain. Additionally, we found that the custom HSI metamodel we designed forms a better fit for the HSI domain than the PREEvision metamodel does.
Preface

This project was conducted within the Software Engineering and Technology group of the Computer Science department of the Eindhoven University of Technology and serves as the result of my graduation project for my Master of Computer Science and Engineering study. It was supervised by dr. Ramon Schiffelers and executed in the form of an Internship at ASML in Veldhoven.

My thanks go out to all people who made this project possible. First and foremost I would like to thank Ramon for his supervision and input during the project. He has provided guidance all throughout the project and was involved at every step. Working with Ramon at ASML was a great experience. Next, would like to thank Yanja Dajsuren for her tutoring during the project. Furthermore I would like to thank prof.dr.ir. Mark van den Brand for global supervision of the project, and dr. Ivan Kurtev for his insights and allowing me to query his expertise. Also I thank dr.ir. Jeroen Voeten for proofreading my work, as well as the members of the assessment committee dr. Aleksander Serebrenik and dr. George Fletcher.

Finally, my thanks go out to my girlfriend Suzanne, my friends, and my family who have supported me throughout the past years.

Frank Razenberg
Contents

List of Figures

1 Introduction
   1.1 Context .......................................................... 2
   1.2 Goals ............................................................ 4
   1.3 Approach ........................................................ 4
   1.4 Outline .......................................................... 5

2 Preliminaries
   2.1 Process Control Applications .................................. 7
   2.2 CARM 2G .......................................................... 8
   2.3 Hardware Software Interface .................................. 10
   2.4 EAST-ADL ........................................................ 10

3 CARM 2G case study
   3.1 Controller overview ............................................. 13
   3.2 Application ...................................................... 14
   3.3 Logical Platform ............................................... 15
   3.4 PREEvision model .............................................. 15

4 Relating CARM 2G and EAST-ADL
   4.1 Architectural concerns ......................................... 17
   4.2 Application layer ................................................ 18
      4.2.1 Modeling Temperature Controller Application in PREEvision .. 18
      4.2.2 PGAPP transformation to PREEvision .................... 19
      4.2.3 Class attributes in PREEvision .......................... 22
      4.2.4 Consistency Framework ................................. 22
      4.2.5 PGSG transformation to PREEvision .................... 23
      4.2.6 PGWB transformation to PREEvision .................... 24
      4.2.7 DV transformation to PREEvision ....................... 25
   4.3 Logical Platform ................................................. 25
      4.3.1 Temperature Controller Logical Platform in Hardware Architecture ... 26
      4.3.2 Temperature Controller Logical Platform in Logical Function Architecture . 27
      4.3.3 Logical Platform in PREEvision Hardware Architecture ............. 27
      4.3.4 Logical Platform in PREEvision Logical Function Architecture ........... 29
   4.4 Comparison of PREEvision and CARM 2G for TWINSCAN modeling ............ 31
      4.4.1 Architecture ............................................. 31
      4.4.2 Language expressivity .................................... 31
      4.4.3 Development ............................................. 32
      4.4.4 Provisional verdict ...................................... 32

Determining metamodel appropriateness for describing a domain
## CONTENTS

5 Hardware Software Interface .......................... 33
  5.1 HSI in PREEvision .................................. 34
  5.2 HSI structure formalization ........................ 34
  5.3 Realization ........................................... 35
    5.3.1 Technological bridge ................................ 35
    5.3.2 Dedicated HSI metamodel .......................... 39
    5.3.3 Transformation from PVHSI to HSI .................. 41

6 Metric: metamodel appropriateness ............... 43
  6.1 Domain appropriateness ............................. 43
    6.1.1 On ontologies ...................................... 44
    6.1.2 Mismatches .......................................... 44
  6.2 Metric design ........................................ 45
    6.2.1 Approach ............................................. 45
    6.2.2 Additional mismatch categories .................... 46
    6.2.3 Scale ................................................ 47
  6.3 Evaluation ............................................ 47
    6.3.1 PREEvision appropriateness for TWINSCAN domain .... 47
    6.3.2 Fit of PREEvision for HSI domain .................. 49

7 Conclusions and future work ....................... 53

Bibliography ............................................ 55
## List of Figures

1.1 Intended expressivity of EAST-ADL for automotive E/E domain .................. 3  
1.2 Possible scenarios of overlap between the TWINSCAN and EAST-ADL domains . 4  
1.3 Situation outline and schematic overview of project approach. We know the TWIN-SCAN domain can be modeled using CARM 2G, and HSI domain can be modeled in EAST-ADL. In this project research whether the inverse is also true. ........ 5  
2.1 Schematic overview of a thermostat control system .............................. 7  
2.2 Simplified overview of ASML process control applications ................... 8  
2.3 Hierarchical overview of CARM 2G languages and separation of disciplines using Y-chart paradigm ................................................................. 9  
2.4 Schematic 3d representation of separation of disciplines and program synthesis through model transformations ................................................. 10  
2.5 Overview of the EAST-ADL architecture ............................................... 11  
3.1 High-level overview of a heater control network .................................. 13  
3.2 Temperature controller application overview. Modeled in PGAPP language. Arrows represent data flow ......................................................... 14  
3.3 Logical Platform for the temperature controller .................................... 15  
4.1 Two alternatives for mapping the CARM 2G languages to EAST-ADL hierarchical layers .......................................................... 18  
4.2 Overview of the resulting temperature controller model in PREEvision ...... 20  
4.3 Example of a PREEvision consistency rule ......................................... 23  
4.4 Built-in mapping mechanism used to map Logical Architecture concepts to Hardware Architecture elements .......................... 26  
4.5 Schematic overview of first alternative for mapping the Logical Platform to PREEvision layers .................................................. 28  
4.6 Architectural overview of mapping from Application onto Logical Platform .... 29  
4.7 Schematic overview of second alternative for mapping the Logical Platform to PREEvision layers .................................................. 30  
4.8 Architectural overview of mapping from Application onto Logical Platform .... 31  
5.1 Overview of the current HSI workflow ............................................... 33  
5.2 Overview of the current HSI workflow ............................................... 34  
5.3 EMF metamodel of the part used from PREEvision to represent the HSI ...... 37  
5.4 EMF metamodel of the datatypes part used in PREEvision .................. 38  
5.5 Schematic overview of instantiation levels used in the HSI and available in Ecore 39  
5.6 Schematic overview of considered clabjects modeling alternatives ...... 40  
5.7 Overview of the dedicated HSI metamodel ........................................ 41  

Determining metamodel appropriateness for describing a domain vii
6.1 Four different categories of mismatches. Lines represent a mapping between ontology and metamodel concepts, and dots represent domain concepts. Green dots map correctly in their category, while red dots indicate mappings with a violation in their category.
Chapter 1

Introduction

In this chapter we discuss the context of the project, problem description, and the approach we used for this research. Next we give our problem statement and provide the outline of the remainder of the document.

1.1 Context

The complexity of software at ASML increases due to its natural evolution, the increasing number of people working on the software simultaneously and the increasingly demanding requirements. To combat these tougher challenges, a novel engineering method where focus remains at ASML’s core activities is desirable. A trend in the software development industry has been the focus shifting rapidly from computing and algorithmic concepts to more model-driven engineering methods specifically tailored after a domain ontology. An ontology is a data model that formally represents knowledge of concepts within a domain and the relationships between these concepts. Anyone with knowledge of the domain should be able to work with software tailored after a domain, regardless of their technical abilities.

Domain ontologies can be captured in metamodels, which can serve as a language definition for a Domain Specific Language (DSL) [10]. A DSL is a computer language specialized to a particular application domain, as opposed to a broadly applicable general-purpose language (GPL). DSLs achieve a high level of abstraction since they are designed exactly after the domain they represent, without any of the clutter found in general purpose languages that has no meaning for the modeled domain. Thus, making use of DSLs makes modeling far more accessible to those who are not as proficient in software development.

To battle the growing complexity of software for its TWINSCAN lithography platform, ASML built the CARM 2G framework for modeling the process control applications that run on a TWINSCAN platform. CARM 2G is powered by a set of DSLs which provide a high-level abstraction over the physical hardware architecture underlying its execution platform. These languages are targeted specifically at the activity of process control. In process control, a controller drives a set of actuators based on a desired state given by a setpoint and feedback from a number of sensors. An example of a proces control application is a temperature controller. Based on the desired temperature setting (the setpoint) and the current room temperature (sensor), the thermostat (controller) can decide to turn on the heater (actuator).

A similar trend of growing engineering challenges can be observed in the automotive industry. With increasing complexity and criticality of vehicle electronics as well as increasing dependability on these electric/electronic (E/E) systems in vehicle, faults due to engineering mistakes have ever increasing impact, both safety-wise as well as financially/legally. For example, a malfunction-
CHAPTER 1. INTRODUCTION

ASML has expressed interest in the form of a research internship where an analysis is made whether technologies from the automotive industry can be employed to further improve the methods used in process control modeling at ASML. One can imagine that the electronic devices in a car are based on many similar concepts also used in the design of process control systems. We intend to research the amount of similarity between these domains. Specifically, we intend to study the TWINSCAN domain by studying the metamodels of the CARM 2G languages, and we will study the EAST-ADL domain by evaluating the EAST-ADL implementation PREEvision.

When we relate EAST-ADL and CARM 2G, three conceivable scenarios are illustrated in Figure 2.3. The figure on the left shows a scenario where there is absolutely no overlap between the two domains. The image on the right shows the scenario where everything fitting in the TWINSCAN domain is somehow present in the EAST-ADL domain as well. The scenario we are most likely to encounter is the one in the middle where there is a partial overlap between the domains. This would imply that some features in the TWINSCAN domain are describable using CARM 2G, but not with EAST-ADL. At this point we cannot make an estimate whether this shared area consists of 50% of the TWINSCAN domain, 80% or perhaps even only 20%. Our goal for this project is to quantify the quality of this fit.
CHAPTER 1. INTRODUCTION

Figure 1.2: Possible scenarios of overlap between the TWINSCAN and EAST-ADL domains

1.2 Goals

From a given domain ontology and multiple metamodels representing this domain, it is unclear how it can be determined which metamodel forms the best fit around the domain. We wish to learn the appropriateness of metamodels in the contexts of the ASML domains TWINSCAN and HSI. The goals of this project are to answer the following research questions:

1. How do EAST-ADL and the CARM 2G framework relate? Can EAST-ADL be used for modeling TWINSCAN models, and if so, how well does the EAST-ADL metamodel fit around the TWINSCAN domain?

2. How do EAST-ADL and the HSI relate? Can we build a metamodel that has a better fit with the HSI domain?

3. How can we, in general, quantify the appropriateness of a metamodel for a given domain?

1.3 Approach

An overview of the current situation and project goals is illustrated in Figure 1.3. We see here that TWINSCAN is modeled using the CARM 2G framework, while the HSI domain is modeled using EAST-ADL. We examine whether TWINSCAN could be modeled using EAST-ADL, and if so, what quality such models would have. We also determine how suited EAST-ADL is for the HSI domain, and attempt to find out which parts of the HSI can be modeled using CARM 2G.

For the TWINSCAN domain a set of domain specific languages from the CARM 2G framework are used to define models in. In order to examine the fit of the EAST-ADL metamodel for CARM 2G, we related both at two different levels of the OMG’s Meta-Object Facility (MOF) [2]. The first step takes place at the M0 level of MOF, i.e. the model-instance level, where we perform a case study by modeling a CARM 2G implementation of a temperature controller in EAST-ADL. We then move up one abstraction layer to the M1 level, i.e. the metamodel level. The case study provides intuition to devise a mapping that allows the modeling of any CARM 2G model as an equivalent EAST-ADL model defined in the EAST-ADL based modeling tool PREEvision.

The choice for modeling with PREEvision stems from the fact that for the HSI domain, models are currently defined in PREEvision. The ASML the Electronic Development division (EDEV) first proposed a structure for the HSI and later realized this structure in PREEvision. We examined how appropriate PREEvision is for modeling the HSI domain. PREEvision is not a generic modeling tool. This means that its underlying metamodel cannot be changed by the user. If some concept is not expressible in PREEvision, the user cannot simply add a class for it to the metamodel. However, PREEvision does allow a tree of generic string-typed (key, value) pairs to be added to any kind of object in the model. This allows virtually any kind of information to
Figure 1.3: Situation outline and schematic overview of project approach. We know the TWINSCAN domain can be modeled using CARM 2G, and HSI domain can be modeled in EAST-ADL. In this project research whether the inverse is also true.

be stored in a model, albeit in a poor way from a modeling and usability perspective. We implemented the structure proposed by EDEV using technology from the Eclipse Modeling Framework (EMF) [20] and built transformations to obtain instances of our custom-tailored metamodel from existing PREEvision models. This clearly exposes the shortcomings of the EAST-ADL metamodel for the HSI. As is, the HSI contains a mixture of information related to both process control applications as well as physical hardware aspects, while only the latter is desirable. Our formalization opens analysis abilities for the HSI, but these fall outside the scope of this project.

In order to quantify how well suited EAST-ADL is for modeling the TWINSCAN and HSI domains, we developed a methodology which captures how well any two metamodels relate. We formalize a metric whose idea is based on work in the area of ontologies. We will evaluate the resulting metric on the metamodels that we have related in the previous goals.

1.4 Outline

The remainder of this document carries the following structure: in Chapter 2 we describe the domains encountered within this project, namely EAST-ADL, ASML’s CARM 2G and HSI. In Chapter 3 we detail the case study of a temperature controller which we performed when relating a CARM 2G model to an EAST-ADL implementation PREEvision. In Chapter 4 we discuss methods that will allow us to define arbitrary CARM 2G models in PREEvision. Then, in Chapter 5 we formalize a proposed metamodel for a domain currently defined in PREEvision. Chapter 6 then combines all previous work to obtain a metric that captures how well two metamodels relate. Finally, we present our conclusions and discuss some future work in Chapter 7.
Chapter 2

Preliminaries

In this project we compare the EAST-ADL architecture description language (ADL) used in the automotive industry, and the CARM 2G framework used at ASML for modeling process control applications in the TWINSCAN lithography platform. Later, we also explore the Hardware Software Interface (HSI) domain which describes available interfaces on a TWINSCAN platform. The following sections provide information about the problem domain areas relevant to this project. These are process control, ASML’s CARM 2G framework, ASML’s HSI, and the automotive EAST-ADL.

2.1 Process Control Applications

One of ASML’s core activities is the design of process control applications. Process control is an engineering discipline that deals with architectures, mechanisms and algorithms for maintaining the output of a specific process within a desired range. A very common process control application is a thermostat as illustrated in Figure 2.1, whose purpose is to maintain a room’s temperature based on a selected preferred temperature. A heater is enabled if the measured temperature according to the input of the room temperature sensor is lower than the setpoint of the desired temperature.

The idea of a thermostat can be generalized to arbitrary kinds of sensors and actuators, and controllers can reach seemingly infinite amounts of complexity. ASML wishes to focus its engineering at modeling control applications and the platform on which these are to be executed. A divide and conquer modeling strategy is desired, where different disciplines can be delegated to different engineering divisions. For facilitate this, an engineering framework is needed with a clear separation between modeling the specifics of a control application, the execution platform, and the deployment of the control application on this execution platform. Such a separation of disciplines facilitates a top-down modeling approach where an overview of the system is formulated first while subsystems can later be refined in greater detail. This approach is illustrated in Figure 2.2.

![Figure 2.1: Schematic overview of a thermostat control system](image-url)
CHAPTER 2. PRELIMINARIES

Since the above does not directly project on the available hardware, an additional special mapping between the abstracted execution platform and actual physical hardware is required.

2.2 CARM 2G

CARM 2G is a framework defined by ASML designed with the goal of enabling analysis early in the multi-disciplinary design process. The motivation for CARM 2G arose from increasing levels of functionality, constant parallel development and an increasing number of engineers working on the same product. The realization of CARM 2G consists of a set of DSLs and a multi-disciplinary development environment to design, analyse and construct control systems for ASML lithoscanners [19]. It allows modeling at different layers of abstraction and decouples the specifications designed by the different disciplines, whilst maintaining clear interfaces. Three modeling layers are used to model application (Application), logical execution platform (LogicalPlatform), and physical execution platform (PhysicalPlatform) respectively, and two layers to map application onto logical platform (AppMap), and to map logical platform components to their physical counterparts in the physical platform (PlatformMapping). Thus, the logical platform is an abstraction of the true physically available execution platform. An overview of the CARM 2G DSLs is shown in Figure 2.3a.

The different CARM 2G languages serve as a separation of domain concepts and their implementation details. Separation of disciplines is achieved by making use of the Y-chart paradigm, or Gajski-Kuhn chart [12], as illustrated in Figure 2.3b. The Application level describes the (control) logic; the Logical Platform level provides the means to implement the logic; and the AppMap mapping level maps each element from the Application level to an element in the Platform level. Within these Y-chart levels, different levels of abstraction of the DSL framework reduce the complexity of the multi-disciplinary parallel design processes [19]. Further refinement of CARM 2G models is achieved through model transformations. This is schematically shown in Figure 2.4. At the top layer we see the CARM 2G languages that describe the process control application, its execution platform and the mapping on a high abstraction level. This is where specification, analysis and synthesis (i.e. worker-block scheduling) is performed. The layer below is a graphical representation of the same system, but now of its realization/implementation.

The CARM 2G framework is implemented as a modeling environment with at its core a set of DSLs defined using the Ecore meta-metamodel [20]. This is the Eclipse realization of the Essential
CHAPTER 2. PRELIMINARIES

(a) CARM 2G architecture overview

(b) Separation of disciplines

Figure 2.3: Hierarchical overview of CARM 2G languages and separation of disciplines using Y-chart paradigm
CHAPTER 2. PRELIMINARIES

Figure 2.4: Schematic 3d representation of separation of disciplines and program synthesis through model transformations

Meta-Object Facility (E-MOF) standard [2]. As such, all features such as code generation, model editors, scheduling algorithms, etc. are implemented using facilities provided by the Eclipse Modeling Framework. Customized editors are used to edit and create models conformant to the DSLs.

2.3 Hardware Software Interface

The Hardware Software Interface (HSI) refers to all interfaces that software on an ASML hardware platform may see or use while running. This includes concepts as sensor readouts, actuator setpoints or power controls of any kind of component. Unlike the CARM 2G framework, the HSI knows no different abstraction levels. All information is specified in one single level, from high-level interfaces up to detailed register mappings on IOBoards. Traditionally, the HSI used to be a large set of textual documents, Visio diagrams and Excel spreadsheets.

An effort has been made by the ASML EDEV division to obtain a more precise definition of the HSI by structuring all this information. This information is modeled using PREEvision, i.e. one of the tools based on EAST-ADL.

Recall that for the CARM 2G languages we will assess whether EAST-ADL can replace them. Since PREEvision is already used to model the HSI, we also know that EAST-ADL is capable of describing the HSI domain, but we cannot specify the quality of the resulting PREEvision models. In Chapter 5 we therefore attempt to assess how suited EAST-ADL is for modeling the HSI. Additionally, since the HSI can function as a rich source of information from which CARM 2G models can be reconstructed, we built the technological foundations needed for extracting information from PREEvision to the CARM 2G framework.

2.4 EAST-ADL

EAST-ADL is described as an Architecture Description Language (ADL) for automotive embedded systems [6, 7, 17]. An ADL is an information model that captures engineering information in a standardized way. In the past, architectures have mostly been defined over a span of documents including a mixture of figures and text in natural language. This complicates communication, changes, validation and has no means to enforce consistency between the several documents. A
standardized notation for representing architectures, i.e. an ADL, attempts to alleviate these issues.

The need for EAST-ADL arises from the increasing complexity and criticality of vehicle electronics. As dependability on electric/electronic (E/E) systems in vehicle increases, faults due to engineering mistakes have ever increasing impact, both safety-wise as well as financially. EAST-ADL offers a well-founded engineering method of describing and implementing automotive embedded systems. A wide range of aspects is covered, including requirements modeling, product variants, modeling of software and hardware components, communication and timing. An overview of EAST-ADL’s architecture is illustrated in Figure 2.5.

Product vendors are free to implement or use a subset of EAST-ADL, tailor chosen aspects to their specific needs, or add features as they desire. The result of this is that EAST-ADL cannot be seen as a true standard. Instead, the EAST-ADL foundation portrays EAST-ADL not as a strict standard, but as a suggested reference. Noteworthy tools claiming to support or be inspired by EAST-ADL are:

- Metacase’s MetaEdit+ [21]: a general purpose commercial tool for modeling languages. A EAST-ADL language implementation is available as a MetaEdit+ profile, allowing one to define EAST-ADL models. This approach enables a clean and efficient way to apply modifications or extend the tool at the metamodel level.

- Eclipse + Papyrus: The EAST-ADL foundations freely available Eclipse-based solution [5] using the Papyrus UML plugin [16] and UML profiles. In a sense this solution can be considered comparable to MetaEdit+, because one can define/adapt the metalevel, and then model using this metamodel. Unfortunately, this implementation has not yet reached maturity.

- Vector’s PREEvision [11, 3]: an EAST-ADL inspired platform able to describe all E/E aspects of vehicles, or according to its vendor, even airplanes. Unlike the MetaEdit+ and Eclipse solutions, with PREEvision we cannot alter the metamodel behind the tool. Although this is desirable, it was chosen to use PREEvision anyway because the EDEV division at ASML is already using PREEvision for the HSI which we discuss in Chapter 5.
The architecture of EAST-ADL consists of four abstraction levels and several extensions as shown in Figure 2.5. It uses a top-down approach where each deeper levels refines the previous. The extensions are defined orthogonal to the extension layers and may therefore reference aspects on different hierarchical levels. At the top-level layer named VehicleLevel the features of a vehicle are modeled in a feature tree. At the next AnalysisLevel allocation of these features is assigned to components. These components describe the abstract functionality of the E/E system. The analysis level is refined by the DesignLevel layer where the functionality of components is made concrete, and where functionality is allocated to hardware components. The final Implementation-Level layer contains the software-based implementation of the system. Finally, several extensions are available that contain constructs that enable modeling of requirements, variability, timing and dependability. None of these extensions are used in this project since they are out of scope for the TWINSCAN domain.
Chapter 3

CARM 2G case study

In this chapter we present the case study of an immersionhood temperature controller which will serve as an example controller that we model in PREEvision. This temperature controller was chosen because it was readily available as CARM 2G models and is easily comprehensible due to its small size yet extensive enough to cover most of the CARM 2G concepts.

3.1 Controller overview

The temperature controller consists of a group of 9 sensors for temperature measurement, a controller, and a heater actuator. The heater control network contains several subsystems as seen in Figure 3.1. In the sensor system the metric temperature signals from the device layer are used as input for the controller networks. Also the sensor filtering takes place in the sensor system.

The temperature setpoint values are constants, since the heaters are used as load compensators. Each of the individual six heaters has their own setpoint. The controller blocks contain 2 main branches: a feedback (FB) branch and a feedforward (FF) branch. The FB part consists of the standard PID controller including the 2nd order Lowpass Filter (LPF).

![Figure 3.1: High-level overview of a heater control network](image-url)
3.2 Application

The CARM 2G implementation of this controller closely resembles the description from the previous section. If we recall the hierarchical overview of the CARM 2g framework, the first layer of interest is the application layer. In this layer we find at the top level language PGAPP, a transducer group instance of sensors that form an input to a servo group instance (controller), which in turn has an output to another instance of another transducer group containing a heater. This is illustrated in Figure 3.2.

The PGAPP (Process Controll Application) language merely contains instantiations of these transducer groups and servo group, and connections between them. Their definitions are given in referenced models in the DV (Device) and PGSG (Process Control Servo Group) languages respectively. Inside the servo group definition we find several connected blocks. Some of these are composite blocks made up of other blocks within, and some are the instantiation of a primary block from the worker block library. These primary blocks have their properties defined in the language PGWB (Process Control Worker Block), while the composite blocks are contained within the same PGSG model.

The transducer groups are defined as models in the language DV. Here, the electrical transducers (ETR) and mechanical transducers (MTR) that make up the transducer group being defined are listed. Currently, their topology does not yet include signal ports and the connections between them. This area is still a work in progress for the CARM 2G framework, so for now it is only possible to reconstruct their hierarchical information. This means we cannot model the type of data flowing at the ports of MTRs and ETRs. As is, ETRs are instantiated solely from their name which can be used to find a complete specification, but this specification is not included in the model. There are plans to add support for the define and instantiate mechanism as is used for blocks inside servo groups.

![Figure 3.2: Temperature controller application overview. Modeled in PGAPP language. Arrows represent data flow.](image-url)
3.3 Logical Platform

The next step is to define an execution platform on which we can map the application. This Logical Platform abstracts from the physical properties of the hardware platform but contains its configuration. Specifically, the logical platform contains *Workers* which abstract from High Performance Process Controllers (HPPC), and *IOWorkers* which abstract from IOBoards. Then, ServoGroups are assigned to Workers and TransducerGroups are assigned to IOWorkers. Last, we need to define connections between Workers and IOWorkers. This logical platform is not part of the temperature controller itself, but defining a compatible logical platform is fairly straightforward.

The logical platform we defined for the temperature controller is rather simple. For each of the two transducer groups we define an IOWorker, and for the servo group controller we define a Worker. Directional connections between these IOWorkers and Worker are then added. A diagram of the resulting model is shown in Figure 3.3.

3.4 PREEvision model

With the application and logical platform of the temperature controller laid out, we obtain an idea of the steps needed to take in order to convert this model into an equivalent one in PREEvision:

- Determine the hierarchical PREEvision layer most suitable for modeling transducer groups and servo groups
- Determine how to model primary block definitions and instances
- Determine how to model composite block definitions and instances
- Determine how ports on and connections between blocks are modeled
- Fill in the remaining gaps, mainly attributes defined on classes in the CARM 2G metamodels

In this chapter we studied an ASML temperature controller modeled in CARM 2G. We looked at the application layer and logical platform of this controller, and determined an approach for modeling the controller in PREEvision. We follow this approach in Chapter 4.
Chapter 4

Relating CARM 2G and EAST-ADL

In this chapter we show how arbitrary CARM 2G models can be transformed to equivalent PREEmvision ones. We first discuss the implications of choices we make with respect to the layered architectures of the CARM 2G languages and EAST-ADL. We then show how the temperature controller discussed in the previous chapter can be modeled in PREEmvision. We continue to elaborate on the precise way in which we map the CARM 2G concepts to compatible PREEmvision counterparts in a generically applicable way. Finally we give a comparison of PREEmvision and CARM 2G, and the implications of their differences for PREEmvision’s suitability for modeling the TWINSCAN domain.

4.1 Architectural concerns

The architectural overviews of CARM 2G and EAST-ADL are very different. Recall the CARM 2G architecture overview from Figure 2.4 which illustrates at the top layer the CARM 2G languages describing the process control application, its execution platform and the mapping on a high abstraction level, and on the bottom layer a graphical representation of the realization of the same system. This architecture obtains separation of disciplines according to the Y-chart paradigm using different domain specific languages for different disciplines. Top-down refinement is obtained through model transformations. EAST-ADL on the other hand does not make use of any such Y-chart paradigm but is instead based on a full top-down refinement approach. If we recall the different architectural layers of EAST-ADL we see that user requirements are modeled as feature trees, which are further specified in the logical analysis and design-levels before their software is finally implemented at the bottom layer. This prompts us to decide whether we adopt the EAST-ADL top-down approach to mimic the CARM 2G languages, or whether we stay true to the Y-chart paradigm by simulating this in PREEmvision. If we decide to adopt EAST-ADL’s refinement model we find that the different CARM 2G languages will map to different EAST-ADL layers (see Figure 4.1a), while if we simulate the Y-chart each of CARM 2G languages map to the same PREEmvision Logical Architecture layer. Since no alternative is clearly favorable over the other, we consider both architectures where we map the different CARM 2G languages in different ways to PREEmvision layers.
4.2 Application layer

The CARM 2G application layer consists of a top-level language PGAPP in which a configuration of transducer groups and servo groups is specified. These transducer groups are defined in the language DV, of which an instance in turn may contain instances of electrical and mechanical transducers. Likewise, servo groups are defined in language PGSG and may contain instances of primary worker blocks as defined in language PGWB. In this section we make explicit how relevant concepts from each language can be transformed into a compatible PREEvision artifact.

The vague use of the term compatible requires some additional detail. This is captured by our goal for this project to establish a methodology to quantify how well some language fits some problem domain which we detail in Chapter 6. To start with we used a model of an immersion-hood temperature controller defined using the CARM 2G languages. In this section we first show how the application parts from this controller can be implemented using PREEvision. We then discuss and motivate the choices we have made during this process, and generalize these choices to extract a transformation between CARM 2G and PREEvision.

4.2.1 Modeling Temperature Controller Application in PREEvision

We first need to determine in which hierarchical layer we model the concepts found in the application layer. From the documentation, “PREEvision’s Logical Function Architecture defines an abstracted graphical function network of the later implementation in software or hardware. The logical architecture encompasses the specifications of logical components such as sensor, actuator and logical functions. In addition, interfaces and connections are specified. Elements of the logical architecture can be linked to the higher customer function layer and the underlying software or hardware architecture.” [11] This description suggests the Logical Function Architecture may be a good fit to model the application layer in.

Additionally, diagrams in the logical architecture can contain instantiations of blocks defined in a shared library, much like the PGWB blocks. It therefore seems a good fit to model the primary PGWB blocks inside this library. In the CARM 2G modeling suite, all primary blocks are defined in a similar fashion where a directory containing definitions of primary blocks is referenced. Inside PREEvision’s design model library we create a new package LogicalTypesPackage containing our logical types. Within here we add new LogicalFunctionType blocks for each primary block we encounter in the CARM 2G temperature controller model.

For example, consider the primary block named MAT_1X9_PGD. From the CARM 2G documentation we find that this represents a 1x9 matrix of PGD values (Process Control Double). This block has 9 input ports and 1 output port. In the PGWB definition of this block we find that each port has a data type PGD assigned to them. These PGD values are double precision floating point numbers with an additional attribute indicating whether this value changes, is valid, and some
event code can be associated with it. This kind of data type can be defined in PREEvision inside a Data Type Package, again in the Design Model Library. In here we define an Implementation Record Data Type which is a composite kind of data type that can contain multiple fields. In here we add a field for the double precision value, two Boolean fields for the changed and valid attributes found in the PGWB type definition, and an integer field for the event code. We can now proceed to add the input port (LogicalReceiverPort) and output ports (LogicalSenderPort) to this logical function type block. Next, we assign a port interface which contains a data element of the PGD type we just defined. This is again done in PREEvision’s shared library, inside another Logical Types Package which, for convenience, we name “Interface Types”. In here we add a new SenderReceiver Interface, which represents an interface that can be assigned to a port in a logical function block. Now, to specify the kind of data that flows over this port we add a Data Element to this interface and specify that its attribute Implementation Data Type points to the PGD type we just defined.

This mostly covers the process of defining blocks and their ports found in the CARM 2G library, but we still need to address a few concepts specific to such blocks. Of primary concern is the ability to specify properties of these blocks. In the case of our matrix block, we still need a way to define the matrix internally stored in this block that is used to multiply the input values with in order to obtain the output value. PREEvision does not offer a suited method to store this kind of matrix on Logical Function blocks. We encoded this information using Generic Attributes. These are nodes of string-typed (key, value) pairs that can be attached to any node in the PREEvision model. We defer discussion of this approach and how it is generically applied to Section 4.2.3.

When all primary blocks found in a full CARM 2G model are defined in the PREEvision library, we can proceed to create an actual controller using these blocks. Inside the Logical Function Architecture node of the model, we add a new element of type Building Block for each of the transducer- and servogroups found in the CARM 2G model. We then add diagrams to these building blocks which we can use to drag blocks onto the canvases of these diagrams. The primary PGWB blocks that we already defined can be dragged from the library onto the canvas, causing them to be instantiated from their logical type. The composite blocks, in CARM 2G named HBG (Hierarchical Block Group), can be modeled using, again, Building Blocks. These blocks may contain inner primary blocks, or deeper nested composite blocks. The inner composite blocks are automatically assigned a new diagram. When finally the full structure of composite and primary blocks is defined in its containing Building Block, we can proceed to define the connection between all blocks. Unconnected ports inside a composite block automatically appear as ports on the block at its container. Connections are made simply by drawing a line between ports that need to be connected. The type of data transmitted over the connection is inferred from the interface assignment of each port.

The PREEvision model obtained from the process outlined above looks as shown in Figure 4.2. Guided by this example, we discuss in the following subsections how each of the elements from the PGAPP, PGSG and PGWB languages can be modeled in PREEvision.

### 4.2.2 PGAPP transformation to PREEvision

We have seen that the CARM 2G PGAPP language contains instances of transducer groups and servo groups defined in PGSG, PGWB and DV models. The PREEvision Logical Function Architecture defines an abstract graphical functional network of the later implementation in software or hardware. This Logical Function Architecture is a seemingly good fit for defining the CARM 2G concepts of the PGAPP, PGSG, PGWB and DV languages in. Additionally, Logical Architecture elements can later be mapped to a Hardware Architecture which could be a good method of specifying how the application platform should be executed on hardware, i.e. the CARM 2G AppMap language.

Summarized, the CARM 2G elements we may find at the root of a PGAPP model are instances
of servo group definitions, instances of transducer group definitions, and connections between them. Several properties can be set on servo group instances, i.e. their name and sampling frequency, and different children can be added to these instances. The following hierarchical tree covers those that are relevant within the scope of this study. The servo group itself is modeled as a PREEvision Building Block which is the composite block element found in the Logical Function Architecture.

In the listing below we show the elements that may appear as the root node of a CARM 2G PGAPP model, and how they are connected to this Building Block representing this root, i.e. how we model these CARM 2G concepts in PREEvision.

- **Servo Group instance**
  A servo group instance is the instantiation of a servo group definition defined in a referenced PGSG model. This enables simple reuse, which is closely mimicked by the PREEvision concepts of Logical Function Types for definitions and Logical Functions as instances of such Logical Function Types.

  - **Servo group definition**
    This property of a servo group instance node contains a reference to the servo group defined later on. Dragging a servo group prototype on a PREEvision diagram’s canvas will instantiate this prototype and internally stores its defining block along with it.

  - **Attributes**
    * **Sample Frequency**
      Every servo group has its own sample frequency. All its blocks are synchronized with this sample frequency.
    * **Set Point Delay**
      Floating point value indicating the delay at which the set point should be recomputed.
    * **Type**
      Enumeration of different types control, controlSimulation, mmdc and mmdcSimulation.

    We discuss in the next section how such attributes, and in fact, every kind of simple attribute can be constructed in PREEvision.

- **Ports**
  Ports are of one of three types:

  - The first type is a dataPort over which data is transmitted. These pass information at every sample, and are typically used for servoloop signals. The value on such ports is
continuously sampled at the frequency at which the controller operates. This kind of port is represented by a sender/receiver port in PREEvision.

- Next is the `injectPort` which is used to inject signals in the servo loop for diagnostic purposes. PREEvision knows no corresponding counterpart that mimics this behavior, but since we focus only on PREEvision’s representational expressivity, we can instead choose to model the `injectPort` as another `dataPort`, where we add an internal attribute indicating this port is in fact a `injectPort`.

- The last type of port is the `eventPort`. Typically these do not pass information at every sample but only once in a while. Their behavior can be simulated by a `dataPort` where an additional Boolean field indicating whether the event is ‘active’. Since no direct PREEvision equivalent for this kind of port is readily available, this motivates how we used sender/receiver ports to model `eventPorts` too.

**Properties**

Values of properties on a block are set when the block is created on a worker and cannot be changed afterwards at runtime. These can again be modeled as described in the next subsection. PREEvision has no mechanism that would prevent these values from being changed after their initial initialization.

**Control mode**

- **Parameter sets**

  A block’s parameters on the other hand can be changed at runtime. Predefined sets of parameters are loaded through some file-based format. All parameters require default values. The use for this is to be able to instantly swap the configuration of some device, i.e. from a high-speed, low precision setting to a low-speed, high-precision one.

  We chose to model such parameter sets using Generic Attributes. We add a Generic Attribute node with name `ParameterSets` to the servo group instance. For each parameter set we add another Generic Attribute node with name `parameterSetX`, where `X` increments with every extra parameter set. The settings in this set are then encoded as key-value pairs stored in Generic Attributes.

  The active parameter set is then encoded as another Generic Attribute `activeParameterSet` of the servo group block, containing as value the name of the active parameter set. This approach has some obvious downsides. Primarily, we have no way of ensuring every kind of parameter receives a value, and there is no guarantee of the validity or even typedness of the values entered by the person modeling the parameter set. Some consistency can be enforced through PREEvision’s consistency framework which we discuss in Section 4.2.4.

- **Metadata**

  A small set of key-value pairs storing some metadata of the servo group instance. These can again be modeled using generic attributes as discussed in the next subsection.

**Transducer Group instance**

- **Definition**

  A transducer group instance describes the instantiation of a transducer group defined elsewhere, namely in a model conformant to the DV language discussed later in this chapter. Transducer group definitions share some features of servo group definitions, meaning we model these again as logical function types in the logical architecture.

- **Type**

  A transducer group consists either of actuators or of sensors, so we add an enumeration type to transducer group instances. In the next subsection we discuss how these are modeled in PREEvision.
CHAPTER 4. RELATING CARM 2G AND EAST-ADL

- **Connection**
  Whereas the class representing all kinds of connections between different blocks for CARM 2G is defined in a core metamodel containing the basics shared by several languages, each connection kind in PREEvision is determined by the kinds of blocks that this connection ties together. In the logical function architecture, these connections are of the type *Logical Assembly Connector*.

- **Attributes**
  All CARM connections have a flag attribute `delaySample` specifying the frequency at which this connection should be read. We add attributes to connections using the method described in the next subsection.

  - **Meta data**
    Similar as the delaySample flag, we add the flags cyclicCut, criticalW2W, znW2W to the *Logical Assembly Connector*.

In PREEvision, ports come in many different kinds, to be used in different hierarchical layers of the model. In CARM 2G, only one kind of generic port is used. Additionally, in PREEvision, an interface can be assigned to a port specifying the kind of data flowing over this port. This is not found in CARM 2G, but would make for a nice addition as it ensures ports are compatible before connecting them.

With the above, we can start transforming applications of CARM 2G models to PREEvision using a top-down approach, i.e. one where we start by taking instances of yet to be defined servo- and transducer groups. A stub for their definitions is automatically inserted, and ports and such on the instantiation are kept in sync with the definition that is to be modified at a later stage. This top-down approach is not possible in CARM 2G. In PREEvision it is possible also to follow a bottom-up approach where, when these servo- and transducer groups are already defined, their instances are added to the application diagram.

4.2.3 **Class attributes in PREEvision**

Many of the basic class properties that are trivial to define in Ecore diagrams and which are frequently used in the CARM 2G metamodels have no direct counterpart in PREEvision. Using the EMF framework, adding such an attribute or reference is simply done by adding it to any class and specifying the type of the data or reference stored in this property. The generated code and editors will automatically ensure type validity for this attribute. It is our desire to model such attributes in PREEvision in a precise manner, but PREEvision’s usability for this is poor. One option offered is to make use of the ability to add arbitrary (name, value) string pairs to any kind of artifact found in a PREEvision model through *Generic Attributes*. Downside of this is that these attributes need to be added on a per-artifact basis, there is no strong-typing available for these, and they can be omitted or, say, misspelled without the model maker even noticing.

4.2.4 **Consistency Framework**

A somewhat stronger way of enforcing the existence and validity of mandatory properties or attributes is by employing the PREEvision framework for defining and checking against consistency rules. These allow the user to describe inconsistencies as a pattern of artifact classes, having specific attribute values and being connected by defined relations. A graphical user interface is available to define such inconsistencies. This is best illustrated by example. Consider the rule shown in Figure 4.3, which requires from Logical Function block types representing *Electronic Transducer Classes* (ETC’s) of the HSI that their name follows a valid format. ETC’s are identified as LogicalFunctionBlockTypes that have a general attribute with name `GENERAL` attached to them, which in turn contains a generic attribute specifying that its classifier has value value
CHAPTER 4. RELATING CARM 2G AND EAST-ADL

Figure 4.3: Example of a PREEvision consistency rule

ETC. For such logical function blocks, unless there is a (Do-not-check, NamingConvention) pair attached to the GENERAL generic attribute, the name should follow the format given in the upper rectangle in the illustration. Defining such consistency rules is cumbersome and executing them takes considerable amounts of processing time, but this does provide a mechanism of enforcing more validity constraints on models. When modeling CARM 2G attributes in PREEvision, consistency rules should be defined wherever sensible.

4.2.5 PGSG transformation to PREEvision

The second layer of the CARM 2G Application platform is the PGSG language, which is used to describe servo groups. Their definitions are stored in a repository which is referenced by PGAPP models. One of the key concepts of servo group definitions is that they can be instantiated. This define and instantiate paradigm is only found at the Logical Function Architecture in PREEvision, where prototypes of Logical Functions, i.e. blocks, are stored in a model-wide library, which can then be instantiated in the logical architecture itself. Since this hierarchical layer is the only one where instances of user-defined blocks are possible, any other hierarchical layer in PREEvision is excluded from being a candidate to model the PGSG concepts in. As such, neither considered architecture as discussed in Section 4.1 differs with respect to the application platform languages.

Servo group definitions contain a combination of primary worker blocks (as defined in the PGWB language, see next section), and hierarchical block groups (HBGs) which in turn consist of worker blocks or again HBGs.

The listing below covers the relevant CARM 2G aspects and how to model these in PREEvision:

- **PGWB Block instance**
  These represent an instantiation of a referenced block from the CARM 2G library. In PREEvision we represent such instantiations as Logical Functions of the corresponding PGWB definition, which is defined as a Logical Function Type in the PREEvision library.

    - Port instances: these represent ports on the referenced block. These can be defined at
the PGWB definition, but omitted at the instantiation. In CARM 2G, all ports used need to be referenced separately, while in PREEvision they are automatically added but can be hidden from the instance without affecting the prototype.

- **Hierarchical Block Group**
  These represent composite blocks which in turn may contain deeper nested HBG’s, port instantiations, connections etc. We have seen already in the previous section that the PREEvision variant for such composite blocks, in our case the servo group itself, is the BuildingBlock artifact. The only structural difference between HBGs and servo group definitions is that only the latter may contain On/Off Block Groups. This restriction has been added as a consistency rule, which we will discuss more thoroughly below.

- **Ports and connection**
  Ports are inherited from the PGWB blocks contained in the servo group. Since we are still working in the Logical Function Architecture, ports and connections are modeled in precisely the same way as described for the PGAPP language in the previous subsection.

- **On/off Block Group**
  The CARM 2G concept of on/off block groups forms a simple method of toggling whether a group of blocks is enabled. Grouping of elements in PREEvision is used for variant management. Different grouped elements are encoded as generic attribute trees.

The root node of a servo group model itself can be seen as another Hierarchical Block Group. As such it is represented as a BuildingBlockType. It is to be instantiated in a Logical Architecture System Diagram, an overview diagram containing and connecting servo groups and transducer groups.

### 4.2.6 PGWB transformation to PREEvision

The PGWB language is used to describe the worker blocks whose instances are combined to form a servo group. Their definitions are stored in a repository which is referenced by PGSG models. In PREEvision, prototypes of logical functions can be stored in a library. Instantiations of these prototype can then be added to diagrams in the logical function architecture.

The listing below shows an hierarchical overview of concepts found in the PGWB language, and a description on how to model these in PREEvision.

- **PG Prim Block**
  These represent a definition of a block from the CARM 2G library. In PREEvision we represent such definitions as *Logical Function Types* in the PREEvision library.

  - **Type declaration**
    A type declaration introduces a type that can be assigned to one or more ports. A type is either a primitive one, a struct consisting of multiple fields with each their own type, an array, an enum, a type variable or a function variable. Primitive types are available in PREEvision as *Implementation Data Types* in the *Data Types* package rooted under the library. Structs consisting of multiple fields with each their own type can be modeled as an *Implementation Record* in PREEvision. Enumeration types in PREEvision are listed under *Common Data Types*. PREEvision has no equivalent for storing function types and types of types.

  - **Port definition**
    PGWB blocks usually have a number of input and output ports. These ports are in fact the same ones as we have seen in the PGAPP and PGSG languages, but they are always defined originally at the PGWB block level. Ports of PGWB blocks that are not
connected within their containing servo group will have these ports available as outputs on the outer building block representing the servo group.

- **Properties and parameters**
  Properties and parameters can be added to PGWB blocks in sets. Parameters are control mode related whereas properties are not. At the PGAPP level, parameters are given a value through a configuration file describing the values of those parameters.

- The attributes with Boolean type IsCGWrapper, IsDVWrapper, IsTimeCritical and PrePostCalcAvailable can be added using the method described in section 4.1.1.

### 4.2.7 DV transformation to PREEvision

The DV language is used to model transducer groups in. These are simpler than servo groups because no process control logic is embedded in them. Additionally, the definitions of MTransducers and ETransducers is not contained in the DV model but instead loosely coupled by merely their name, unlike for the PGSG language where the block instantiations are actually further detailed in the PGWB language.

The listing below covers the DV aspects and how to model these in PREEvision:

- **Transducer group definition**

  - **Type attribute**
    This attribute specifies whether this is a group of sensors or actuators. Refer to section 4.1.1 for a generically applicable way to model these string and enumeration attributes.

  - **ETransducer and MTransducer definitions**
    ETransducers and MTransducers are not modeled in detail, just their name and ports are stored in the CARM 2G model. Since every PREEvision artifact has a name, we can simply use an empty logical function block with this name.

    - **Ports and connections**
      Since transducer groups are modeled in the PREEvision Logical Function Architecture, the same applies to the modeling of ports and connections as for the ports and connections as found in the PGSG and PGWG languages.

        Additionally, we set the interface type of each logical sender or receiver port to a custom type `DV_Port`. Sensor input filters and actuator output filters must therefore also be assigned this interface type.

In PREEvision we wrap the transducer group in a Building Block Type containing the logical functions describing the E- and MTransducers contained in the group. The Building Block Types are added to the PREEvision library and instantiated in a Logical Architecture System Diagram, along with the Building Block Types describing the servo groups to be instantiated. Connections between the transducer and servo groups are then drawn by connecting the corresponding ports, which should all have an interface assignment of `DV_Port`.

### 4.3 Logical Platform

Despite its apparent simplicity, modeling a logical platform in PREEvision actually requires a fair bit of additional thought. As with the application platform, we will first need to decide which of PREEvision’s logical layers to use. Choosing PREEvision’s software or hardware architecture layer to model the logical platform in enables the use of its built-in mapping mechanism for defining a mapping between logical components and their hardware or software implementations. Then again, the only artifacts that are readily available to be placed in these layers are the only
CHAPTER 4. RELATING CARM 2G AND EAST-ADL

Figure 4.4: Built-in mapping mechanism used to map Logical Architecture concepts to Hardware Architecture elements

ones that can be used, unlike in the logical architecture, where we could instantiate prototypes that we defined ourselves in the shared library. The idea of defining your own prototypes and instantiating them is referred to as *ontological instantiation* in literature, whereas making use of language-provided concepts is named *linguistic instantiation* [2, 15]. This means that choosing either this hardware or software architecture level would sacrifice a lot of flexibility. We decided therefore to also consider modeling the logical platform in the Logical Function Architecture as well. We will discuss either alternative, and their implications for the deeper CARM 2G layers.

4.3.1 Temperature Controller Logical Platform in Hardware Architecture

The PREEvision hardware architecture is fairly simple. It is divided in three parts of increasing level of detail. First is the *hardware components and network topology*, where, as the name suggests, the configuration of high-level hardware components is arranged. Next are the electric circuit and wiring harness. These focus on electrical aspects and physical representation of wires, cables and splices. Only the first part is of interest to us as this already suffices to model the small CARM 2G logical platform in.

The ECU (Electronic Control Unit) component available in the hardware architecture’s component diagrams is used for controlling and regulating the functionality of the E/E architecture. Effectively, its intended use is not as important as the representational purposes that we have in mind for it. To such an ECU component we can add process units. The built-in mapping mechanism allows components in the logical architecture to be assigned to process units on an ECU, representing that this particular process unit is supposed to carry out the functionality provided by the mapped logical function component. This seemingly resembles the Worker and IOWorker concepts of the CARM 2G logical platform, where servo groups and transducer groups are mapped onto. The CARM 2G metamodel also facilitates mapping hierarchical blocks within servo groups to processing units of Workers.

In PREEvision mapping a logical function directly to an ECU is not possible, a process unit is required for that. This restriction can be overcome easily by adding a single process unit to every ECU that is dedicated solely for servo groups to be mapped to. Adding this process unit on may automatically be performed if we define a propagation rule for this. Another restriction is that on a simple ECU, no distinction is made between Workers and IOWorkers, resulting in the possibility of assigning servo groups to IOWorkers instead of Workers. This can be prevented by forbidding this using consistency rules. Both propagation and consistency rules will be explored in Section 4.2.4. An example of a possible resulting mapping is given in Figure 4.4.

Another consideration that should be taken into account is the ease of mapping this logical platform to the physical platform. The only available PREEvision mapping layer sourcing from the hardware architecture is toward the *geometry* layer, which has no purpose for the CARM 2G domain. This means a proper mapping between logical and physical platform is at least somewhat complicated. One way of dealing with this is to move the logical platform up to the software architecture of PREEvision. This has primarily the advantage that the built-in mechanism for mapping software components to hardware components is possible if we could then map the
physical platform in the PREEvision hardware components layer. Additional downside of this approach however, is that no component exists in the software layer that could fulfill the role of processing unit on an ECU. Due to the complication we already encountered, and the additional difficulties we expect, we have not pursued going down this path any further.

4.3.2 Temperature Controller Logical Platform in Logical Function Architecture

The limitations encountered when defining our logical platform in the hardware or software architectures of PREEvision prompt us to look further. Our alternative is to model this platform in the same logical architecture as we modeled the application platform in. This means we define logical function type blocks for Worker and IOWorker, and place these in a diagram separate from the one describing the application platform. We then need to decide how to model processing units, and how to map the servo- and transducer groups of the logical platform onto these Workers and IOWorkers.

Two ways of modeling processing units are possible. The first would be to have ports on the Worker/IOWorker blocks represent a processing unit. This port can then also be used to link to a hierarchical block representing that this block is executed on the linked processing unit. Whether this is a proper way of modeling our logical platform is debatable. For all its benefits, the use of ports in this way is definitely a hack. A second way would be to model workers as a hierarchical block, and have simple inner blocks for its processing units.

What is left is to define a way to model the mappings between servo groups/transducer groups and Workers/IOWorkers, and the mappings between hierarchical blocks and processing units. We have already briefly seen that this can be achieved by using ports for this purpose. A better way would be to define a way where this mapping is made explicit on the application platform components as some additional attribute. Unfortunately, PREEvision offers no proper way of modeling this, but we evaluated a few options in the next sections.

4.3.3 Logical Platform in PREEvision Hardware Architecture

Guided by the temperature controller example and both alternatives to model its Logical Platform in PREEvision, we discuss in the following subsections how a transformation to either alternative can be obtained for arbitrary CARM 2G Logical Platform models.

In the Logical Platform a set of Workers and IOWorkers with connections between these are defined. Servo groups can later be assigned to Workers and transducer groups can be assigned to IOWorkers in the AppMap language. We have seen while modeling the immersion hood temperature controller that, unlike for the application languages, picking a suitable PREEvision layer for the CARM 2G logical platform is not a trivial choice. The alternative described in this section corresponds to the alternative shown in Figures 4.1a and 4.5.

The listing below shows the relevant CARM 2G concepts and a description of how they are mapped to PREEvision artefacts.

• Logical Platform Definition

The PREEvision Hardware Architecture contains a Network Diagram which can contain components that share some resemblance to those found in CARM2G’s Logical Platform. Namely, the ECU component can be given any number of processing units, and the built-in mapping mechanism allows for assigning logical function blocks to processing units on ECU’s in the Hardware Architecture.
CHAPTER 4. RELATING CARM 2G AND EAST-ADL

Figure 4.5: Schematic overview of first alternative for mapping the Logical Platform to PREEvision layers

- **IOWorker**
The ECU component available in PREEvision’s hardware architecture’s Network Diagram is used to represent CARM 2G IOWorkers. This ECU, unlike the Logical Functions we have seen at the application level, is not an instantiation of a user-defined type, but instead an instance of a type defined in the PREEvision metamodel.

In order to be able to distinguish between Workers and IOWorkers, we restrict the name of IOWorkers to start with “IOW_”.

- **Processing unit**
  These come in three types *controlProcessing, signalProcessing* and *backgroundProcessing*. The distinction is made by adding an enumeration *type* as an attribute to every IOWorker.

- **Worker**
  Workers are in most regards similar to IOWorkers. For modeling the Logical Platform it suffices to add that Workers are ECU’s, with the restriction that its name should begin with “W_”.

- **Connection**
  Connections between Workers and IOWorkers. In the PREEvision Hardware Architecture the ports are of type *Conventional Connection* and the ports of type *Conventional Connector*. No additional properties are encoded on these ports or connections in CARM 2G.

**Mapping Application to Logical Platform**
The application as defined in the PGAPP language consists of servo groups and transducer groups. The logical platform defined in the LogicalPlatform language consists of Workers and IOWorkers. A third language *AppMap* allows for mapping servo groups onto Workers and transducer groups onto IOWorkers. Typically, additional information is encoded in the application mapping, such as the task schedule for the control blocks on the computing nodes [19].

We model this AppMap language using the PREEvision built-in mapping mechanism that allows connection logical functions of the logical function architecture to ECU’s in the hardware architecture. PREEvision allows only for mappings between logical functions and processing units on ECUs, while in CARM we can directly associate to Workers and IOWorkers. This requires us to make a dummy processing unit on each ECU with no purpose other than for servo groups or transducer groups to be assigned to. An example such mapping was given in Figure 4.4.
Without additional restrictions on what kind of logical function may be assigned to what kind of ECU, it would also be perfectly valid to assign the building block representing a servo group to a processing unit of an ECU representing an IOWorker, while in CARM 2G servo groups are exclusively assignable to Workers. As such, we need two consistency rules to forbid such assignments. We identify ECUs representing a Worker as those whose name starts with “W_”, and IOWorkers are identified as the ECUs having a name that starts with “IOW_”. Servo groups are those Building Block instantiations whose name starts with “SG_” and transducer groups must have a name starting with “TG_”. Encoding this information in one rule disallowing servo groups to be assigned to IOWorkers and one rule to disallow transducer groups to be assigned to Workers obtains the desired result.

Now, hierarchical blocks inside servo groups may be assigned to process units of Workers. The PREEvision mapping mechanism readily accommodates this. We expand the consistency rule for Workers by stating that an HGB can be assigned to a processing unit on a ECU if this ECU represents a worker and the processing unit is not the dummy one to which only servo groups can be assigned. HBGs are identified as a Building Block Type whose name does not start with “SG_” or “TG_”, as those would be servo groups or transducer groups.

An architectural overview of this mapping from Application onto Logical Platform using our first alternative approach is given in Figure 4.6.

4.3.4 Logical Platform in PREEvision Logical Function Architecture

We evaluated a second way of defining the Logical Platform in PREEvision. Here we define Workers and IOWorkers as blocks in the Logical Function Architecture. This method corresponds to the alternative shown in Figure 4.1b.

- **Logical Platform Definition**
  The logical platform root is placed as a Logical Architecture System Diagram in the Logical Function Architecture. Additionally, the application platform is moved in another such system diagram. This enables proper scoping of both layers.

  - **IOWorker, Worker**
    We use a BuildingBlockType with two special Logical Functions inside to model the Worker. These blocks are used to model the processing units and connection points associated with this Worker. Generic Attribute ‘ecuType’ specifies whether this is an IOWorker or a Worker.
* **Processing unit**
  Processing Units are represented by ports on the inner Logical Function with name `ProcessingUnits`. These shall all be of the `LogicalReceiverPort` type.

* **Connection point**
  Each connection port is mapped to a port on the inner Logical Function with name ‘ConnectionPoints’. Directionality of ports corresponds with the PREEvision defaults, meaning `write` ports correspond to `LogicalSenderPorts` and `read` ports correspond to `LogicalReceiverPorts`.

* **Attributes and metadata**
  Each metadata property receiver a separate generic attribute. These are to be validated using consistency rules.

A consequence of this approach is that each IOWorker needs a unique name in the library. Due to different configurations on the inner blocks, no generic i.e. reusable type can be prepared in the library. The CARM 2G equivalent of Workers and IOWorkers follows a linguistic instantiation which was matched in our first alternative for the Logical Platform. In this alternative however, we instantiate the Workers and IOWorkers from a self-defined Logical Function Type. Although this instantiation occurs just once we observe here an instance of *ontological instantiation*. Furthermore, in order to be able to distinguish between Workers and IOWorkers, we restrict the name of IOWorkers to start with “IOW_”.

**Mapping Application to Logical Platform**

With the CARM 2G Logical Platform encoded in PREEvision’s Logical Function Architecture we can define the application to platform mapping as follows. The AppMap rootnode will be stored as a third Logical Architecture System Diagram in the Logical Function Architecture. On this diagram we drag the servo groups and transducer groups from the application platform, and also the Workers and IOWorkers from the Logical Platform. Note that these will just be references to these blocks, and not copies. Now, mapping of servo groups to Workers can be represented by a connection between dedicated mapping ports on the outer `BuildingBlockType` of the servo group and the inner Logical Function with `ConnectionPoints` inside the Worker. For completeness sake, we assign to the ports the interface `PU_SG_Assign`. Analogously, Hierarchical Block Groups
map to Processing Units of Worker blocks with port interface $P_U\_H_{BG\_Assign}$. Next, Transducer Groups map to IOWorker blocks using interface type $P_U\_T_G\_Assign$, and finally ET and MT instances map to Processing Units of IOWorkers with interface definition $P_U\_H_{BG\_Assign}$.

An architectural overview of this mapping from Application onto Logical Platform is given in Figure 4.8.

4.4 Comparison of PREEvision and CARM 2G for TWINSCAN modeling

Having researched the capabilities of PREEvision we can make a comparison between PREEvision and the CARM 2G framework for modeling problems in the TWINSCAN domain.

4.4.1 Architecture

CARM 2G and EAST-ADL are built on strongly differing modeling approaches: in CARM 2G the Y-chart development approach is centralized whereas EAST-ADL structures models after refinement steps. PREEvision contains no method to properly replicate or simulate the development after this Y-chart paradigm, which is essential to the TWINSCAN domain and the heavily incorporated way of working at ASML. The refinement abilities of EAST-ADL serve little use for use in the TWINSCAN domain.

4.4.2 Language expressivity

We can assume that the CARM 2G languages form a close fit to the problem domain as they were developed precisely to model problems in this domain. PREEvision on the other hand is much broader: besides the concepts already discussed in this chapter, PREEvision also offers features for modeling matters such as feature trees, configuration variants, electric design, geometry and
wiring, administration and even personnel management. The domain PREEvision aims to tackle is much wider than what the CARM 2G languages target, but the modeling requirements for CARM 2G are much more specific.

The result of this is that, as we have seen, all concepts found in CARM 2G can somehow be described in PREEvision due to its ability to store trees of arbitrary (key, value) pairs along with any tree node found in a model. Keeping in mind that PREEvision had to enable modeling of a rather generic domain, we anticipated most CARM 2G concepts could be modeled too, but clearly never expected PREEvision to fit as well as the CARM 2G languages do themselves. In order to solidify this feeling, we need a method to quantify how well-suited PREEvision is for TWINSCAN modeling. We therefore introduced a metric capturing this intent in Chapter 6.

4.4.3 Development
Using PREEvision, a large burden is introduced when coupling elementary information in the form of class attributes to any of the concepts found in the CARM 2G languages. Essentially, simply being able to attach a set of generic attributes to any kind of artifact already establishes enough genericity to model virtually any information as long as the user is imposed restrictions on convention. PREEvision facilitates this using a framework for modeling consistency rules that we deployed for the enforcement of the conventions required. This solution works reasonably well for the modeling of simple attributes such as integers, strings or enumeration values to some content. In fact, consistency rules can even be used to enforce proper referencing between concepts, i.e. between a servo group and the Worker it is deployed on. However, all sense of scoping is lost and each of these consistency rules is entirely dependent on the user at least adhering to the naming of concepts with an identifying prefix. When a referenced concept is renamed, all references to it are invalidated. On the other hand, when using EMF, the effort required to model attributes of classes is minimal, attribute values or references require no additional checking and maintenance is simple.

Using model transformations for the synthesizing of models, text, or code is in CARM 2G enabled with the large set of tools offered by the Eclipse Modeling Framework such as Acceleo and QVTo. Even generated Java code can be used in a Java project, enabling loading, modifying and storing these models. This is another area where PREEvision falls short. The metamodel used to store PREEvision models is not publicly available meaning no interaction with the stored model is facilitated through a stable API.

4.4.4 Provisional verdict
It appears that the facilities provided by PREEvision are not a good fit for the creation of models in the part of the CARM 2G domain we considered. Primary contributors to this are the fact all benefits from using the Y-chart approach are lost, that flexibility is lacking with respect to defining custom attributes on artifacts, and the absence of a convenient method to further synthesize a PREEvision model. For now we have no numerical data to strengthen our verdict. After evaluating the metric proposed in Chapter 6 we draw our final conclusions in Chapter 7.
Chapter 5

Hardware Software Interface

The ASML Hardware-Software Interface describes all interfaces that software may see or use when running on a specific hardware platform. This can include e.g. sensor readouts, actuator setpoints or electronic controls of the platform itself such as the power status of some component. One of the ongoing difficulties of the HSI is the fact that it contains a mixture of information related to both process control applications as well as physical hardware aspects, without any separation in different abstraction levels. In an ideal situation, the HSI would consist only of hardware related concepts.

In order to eliminate undesired information from the HSI, its elements need to be positioned against their counterparts in the CARM 2G language stack. Currently the HSI is encoded as a PREEvision model. In order to enable such a positioning of elements we need a stronger foundation of the HSI’s structure. Therefore we formalize the HSI according to the metamodel proposed by EDEV and encoded in PREEvision by developing an EMF Ecore metamodel of the HSI. The full activity can be illustrated as in Figure 5.1. In this project we intend to take care of the formalization process. The positioning of HSI concepts to the CARM 2G languages is scheduled for 2014.

In this chapter we first describe how the HSI was modeled in PREEvision. The remainder is devoted to the realization of our metamodel design and the transformations we implemented to map a PREEvision HSI instance to an equivalent one structured after our improved metamodel. This additionally enables a comparison using our metric detailed in Chapter 6 between the PREEvision metamodel, and the custom-tailored metamodel that specifically targets the HSI, allowing us to quantify how well PREEvision is suited for modeling the HSI, or rather, how much improvement is possible when using a metamodel specifically tailored after the HSI.

Figure 5.1: Overview of the current HSI workflow

Determining metamodel appropriateness for describing a domain
CHAPTER 5. HARDWARE SOFTWARE INTERFACE

5.1 HSI in PREEvision

Traditionally, the HSI was defined as a poorly organized set of Visio diagrams, Excel files and other documents. The Electronic Devices division (EDEV) at ASML has managed to combine the information from this collection of files and structure this in a single PREEvision model. A PREEvision HSI model is obtained from the unstructured files by running an importer on them. This importer is defined as Java code within PREEvision. In PREEvision, the user can write snippets of Java code to perform complex computations to calculate metric reports on (parts of) the PREEvision model. These metrics capabilities inside PREEvision allow direct access to the model elements from Java code, which is used as an entry point to the model for all importer and exporter functionality.

Since the PREEvision metamodel was not specifically tailored to model precisely the structure of an HSI in, it makes sense that a 1 to 1 mapping between HSI elements and PREEvision artifacts is not possible. Therefore, to be able to distinguish between different items that are mapped onto the same PREEvision artifact, EDEV made extensive use of conventions paired with Generic Attributes. To illustrate this by an example, take the ASML Class/Type/Instance hierarchy paradigm which differs from the common object oriented Class/Object hierarchy in the sense that Types are a refinement of a Class with some default configuration of the class’ attributes whilst possibly restricting visibility of these attributes to instantiations of this type. In literature this ‘type’ construct is typically referred to as clabject. An instance is then instantiated from such a type. Recall that PREEvision does support a Class/Instance hierarchy natively through a Logical Function instantiated from a Logical Function Type, but there is no built-in support for this Class/Type/Instance hierarchy. As such, EDEV chose to model both Classes and Types as Logical Function Type artifacts, where distinguishing between both is done by assigning a generic attribute ‘TYPE’ with value ‘CLASS’ to the Logical Function Types that represent a class.

5.2 HSI structure formalization

The part of PREEvision’s metamodel that is actually used to model the HSI in consists of only a small fraction of the entire PREEvision metamodel. We have seen while relating CARM 2G and PREEvision that modeling of typed attributes in PREEvision is cumbersome and no truly desirable solution can be obtained. Yet, the HSI consists largely of relatively simple objects with lots of typed attributes. One can imagine that in hindsight, PREEvision may not have been the appropriate tool to model the HSI in. As an addendum to this graduation project, it was decided that we lay the foundations for an alternative way to model the HSI using techniques from the Eclipse Modeling Framework.

Currently, the workflow starts with a set of unstructured documents. These are imported into a PREEvision model that can be understood when read together with textual descriptions that provide an understanding of the PREEvision model. Other than the Java importers defined as PREEvision metric calculating snippets, EDEV also defined several exporters. These output XML files in a format that can be compiled into deployable binaries that run on the hardware platform. Like the importers, these exporters too are implemented using Java code in PREEvision metrics.

Figure 5.2: Overview of the current HSI workflow
CHAPTER 5. HARDWARE SOFTWARE INTERFACE

It is desirable to obtain a better, self-describing formalization of the HSI. We believe that building a custom-tailored metamodel for the HSI would realize precisely this. It is our aim for this part of the project to build such a metamodel, and define the necessary transformations to obtain instances of these models from existing PREEvision HSI models.

5.3 Realization

In order to realize the desired formalization of the HSI, we observe three initial steps that need to be taken. First a technological bridge from PREEvision to EMF needs to be devised. This allows us to further transform the HSI model using the powerful EMF techniques, in particular the QVTo model transformation language. The next step is to actually build a metamodel that is modeled after the actual HSI structure. Finally, we want to automate the conversion from PREEvision HSI models to instances of our improved metamodel.

5.3.1 Technological bridge

The first step to take in the process of the HSI formalization is to build a technological bridge to the EMF toolset. This requires us to do two things: 1) build a EMF metamodel to hold precisely the artifacts stored in the PREEvision HSI models, and 2) develop a transformation from PREEvision HSI models to our compatible EMF metamodel.

As mentioned before, the actual part of the PREEvision metamodel used to hold HSI models is relatively small. Whilst this is unfortunate from a modeling perspective due to the additional required restrictions on convention within PREEvision, this does limit the effort required to fulfill this step. We have developed a compatible metamodel from the textual descriptions provided by EDEV and from inspection of existing PREEvision HSI models. This cloned metamodel is mostly a straightforward copy of the relevant parts of the PREEvision metamodel. Additionally, we have built the PVHSI EMF metamodel in a somewhat simpler way by making some generalizations. To illustrate this, consider some of the abstract metamodel classes of the PREEvision metamodel such as $M\text{HasGenericAttribute}$ and $M\text{NamedArtifact}$ indicating respectively whether an artifact can contain generic attributes and whether the artifact has a name. Since in fact every PREEvision artifact that we encounter in the HSI model inherits from both these abstract classes, we decided to introduce in our model a single abstract base class $artifact$ which covers the functionality from both PREEvision’s $M\text{HasGenericAttribute}$ and $M\text{NamedAttribute}$ classes. This simplifies our Java transformation and results in a smaller and more comprehensible metamodel. This resulting metamodel is shown in Figure 5.3 and will hereafter be referred to as $EMF\ PVHSI$ metamodel.

PREEvision offers exhaustive possibilities to model and structure different kinds of data types. Since a large part of the offered functionality w.r.t. these data types is actually used by the EDEV HSI models, we chose to extract this part of the metamodel to a separate one that we will directly incorporate with the EMF HSI metamodel. The resulting metamodel for these datatypes is given in Figure 5.4.

The EMF toolset facilitates automatic generation of classes defined in this metamodel, as well as methods to load and persist such models. In order to transform existing PREEvision HSI models to instances of our EMF metamodel we make extensive use of this generated API. We wrote Java code inside a PREEvision metric snippet to accomplish this. This code makes use of the EMF API and generated code from our metamodel.

Developing this transformation mapping is straightforward as for every PREEvision artifact that we encounter we have added a matching counterpart in the target metamodel. Still, the con-
version is somewhat cumbersome as we need to manually traverse the entire PREEvision model in Java code, instead of being able to use the niceties offered by a true transformation language such as QVTo. The resulting Java code is wrapped inside a PREEvision metric snippet and was successfully tested to clone HSI models by exporting them to instances of our EMF PVHSI metamodel.
DETERMINING METAMODEL APPROPRIATNESS FOR DESCRIBING A DOMAIN

Figure 5.3: EMF metamodel of the part used from PREEvision to represent the HSI
Determining metamodel appropriateness for describing a domain

Figure 5.4: EMF metamodel of the datatypes part used in PREEvision.
5.3.2 Dedicated HSI metamodel

Having prepared a transformable representation of the original PREEvision HSI model we can move on to developing a better metamodel whose structure more directly accommodates that of the HSI. This is arguably the hardest step and biggest effort in the formalization process. The HSI can be split up into a small set of distinctive entities, namely:

- ASML Types (ETransducer, MTransducer, ParameterGroups, etc.)
- Types library
- EDEV Release
- Application Specific Board
- Electronic Board
- IO Cluster
- Board Map

In the following subsections we discuss these, and how we represent them in our metamodel.

ASML types

All HSI objects have a certain type (MT, ET, BF, EA, GP, PB and PV) and are described in a similar way. In the basis every artefact has a class, type and instance. It was decided by EDEV that classes are modeled as a Logical Type in the Library, types are modeled as a Logical Type in the Library, and Instance is an instantiation of the Logical Type in the Logical Architecture. Possibly the most fundamental decision to be made for our metamodel is precisely how we design our dedicated metamodel to accommodate this. Note that in this paradigm we encounter two different kinds of instantiation, namely instantiation of classes as types, and of types as instances. In Ecore however, only one level of instantiation is available to the model designer, namely from metamodel → model level. A schematic overview can be seen in Figure 5.5.

We need a way to model the different kinds of instantiation in our metamodel. In [22] a method tackling this problem is suggested. One approach requires the introduction of a special kind of reference between objects denoting instantiation. Such changes would have fundamental implications for the Ecore implementation. In [1] Atkinson et al. suggest an approach where a metamodel is generated from information stored in a model. This metamodel is then merged with the original so that a strong typing of deeper layered instantiations become available. This alternative is schematically presented in Figure 5.6b. Since we are restricted to the use of the EMF toolset no perfect solution is available. Instead, two Ecore-based alternatives were researched:

1. Define class, type and instance as EClasses. All instantiations occur in the model. The ontological meaning of instantiation is lost, as class/type-instance are all instantiations of their respective EClass. However, OCL constraints or Java code can be used to assert the ASML type hierarchy is actually valid.
2. Create one metamodel MM1 to contain the ASML classes. An instantiation M1 then contains the types, which, using a model transformation can be transformed into a ‘variable’ metamodel MM2 containing these types. Instances then reference this transformed metamodel. This method clouds the toolchain integration and may therefore not be desirable, but comes with the benefit that type properties will be available at compile time, allowing better editors to be generated because they can now incorporate type information.

Schematically, both alternatives can be described as shown in Figure 5.6.

The downside when choosing alternative 1 is that we lose the ontological meaning of instantiation at the model level. We decided to go with the first alternative because we believe the more fluent toolchain integration outweighs having a true ontological meaning of instantiation at the model level. Unfortunately, the result of this is that we still do not achieve proper typedness of class and type attributes and will instead need to rely on the editors that will be defined for our models to take on these validation tasks.

Library

All available ASML classes and types are stored in a central library. The PREEvision model uses the built-in library facility to store these. Within this library block definitions and data type definitions can be organized in a folder-like structure. For our dedicated metamodel we modeled a similar kind of library. A single HSI model typically contains two libraries; one containing the ASML primitive types, and another containing the composite types used within the hardware platform described by the model. We enable distinction between these by adding an attribute specifying the kind of library to the library class in the metamodel.

EDEV Release

The primary entity described in a HSI model is a full subsystem such as a waferstage or immersion hood. Conformant to the ASML way of working it was deemed the most sensible choice to model complete subsystems as separate models. Any other choice would cause too much defocus. Yet, this choice carries the downside that too much data may be presented to end users, or even data to which users may not have rights. In order to minimize the challenge in modeling the HSI it was chosen to ignore these drawbacks and instead focus on mastering the challenge of employing PREEvision for the purpose of modeling complete subsystems. For this master’s project we have no intentions on improving or changing these aspects. Our focus lies with formalizing the HSI structure proposed by EDEV, not on making alterations to this. We believe this is accomplished by migrating the used technology from PREEvision to the Eclipse Modeling Framework because
of the structure that is naturally encoded in Ecore metamodels.

The deliverable for every subsystem is named an \textit{EDEV Release}. As such, it should come as no surprise that the root node of our metamodel is a class named EDEV Release. We tie the aforementioned libraries, along with IOBoards and IOClusters to an EDEV Release, thereby forming the core of the metamodel. An excerpt is shown in Figure 5.7. Throughout the metamodel we will encounter several classes whose name ends in “Info”. These are specialized configuration structures formatted after the information they need to contain when compiling them by the \textit{SCB} tool. For now, these are modeled in a generic way where there information is stored in a hierarchical tree of name-value pairs. This directly opposes the goal of moving from a “loose” metamodel to a tightly structured one. Regardless, it was chosen not to touch these areas due to the overwhelming scope that would be introduced if we were to properly structure this.

![Figure 5.7: Overview of the dedicated HSI metamodel](image)

Furthermore the EDEV release instances of the Application Specific Board, Electronic Board, IO Clusters and Board Mappings. For these we copied the structure proposed by EDEV while substituting the methods used to define reference to and give valuations of instantiations of the ASML types by our own mechanism for the ASML types. Additionally, these areas contain large chunks of information which we encoded as trees of Generic Attributes, i.e. a more or less one-to-one clone of the PREEvision model representing them. These areas were considered too domain specific and gains obtained from properly structuring these would be negligible, since this data is formatted after its expected export format which is fed into a later compilation step.

### 5.3.3 Transformation from PVHSI to HSI

As a final step in the HSI formalization process we needed to define a mapping from the PREEvision-compatible EMF metamodel, to our custom tailored HSI model. Since we already make frequent use of EMF technology the QVTo transformation language \cite{8, 14} is particularly suited to perform this task in. Somewhat poor documentation, or rather, lack of official documentation makes for a rough start when beginning to write such transformations, but one quickly picks this up due to intuitivity of the language and expressivity of OCL expressions. Additionally, due to the acquired knowledge of both source and target metamodels, the development of this transformation went smoothly. The resulting QVTo program transforms an existing EMF PVHSI model to a more structured instance of our dedicated metamodel.
Chapter 6

Metric: metamodel appropriateness

In the previous two chapters we have looked at two different domains, TWINSCAN and HSI, and for each domain we studied two metamodels used to define domain models in. Specifically, for TWINSCAN we studied the CARM 2G languages which are specifically designed for this domain, and at the more generically applicable automotive modeling tool PREEvision. Next we looked at the HSI domain, how PREEvision was used to define models in the HSI domain, and we developed an EMF metamodel specifically targeted at the HSI’s structure. Our intuition has been that for the TWINSCAN domain, PREEvision is a worse tool to represent TWINSCAN models in than the CARM 2G languages are. For the HSI domain too, PREEvision has limitations that we attempted to resolve by defining a better fitting metamodel. We now wish to confirm our intuition. No existing methodology appears to be available to numerically express the quality of such a fit. Therefore, we developed a metric expressing how well one metamodel fits a domain by relating this to a reference metamodel considered to be a perfect abstraction of the domain under consideration.

In this chapter we first discuss the notion of domain appropriateness as presented by Guizzardi et al. [13]. Next, we explain how we constructed a metric based on domain appropriateness that can be employed in our context, i.e. the context of deciding how well a metamodel fits a given domain. Finally, we evaluate the metric on the metamodels we studied in the previous chapters and discuss our findings.

6.1 Domain appropriateness

In their work An ontology-based approach for evaluating the domain appropriateness and comprehensibility appropriateness of modeling languages [13], Guizzardi et al. introduce the notion of domain appropriateness by observing the level of isomorphism between a perfect domain abstraction and a concrete representation of the same domain. A framework is proposed which enables evaluating the suitability of a language to model real-world phenomena in a given domain. To this end, a formal representation of a conceptualization of some domain, and a concrete representation of the language used to describe that domain, are compared. The formal representation of a domain is typically referred to as a reference ontology, a rich axiomatic theory whose focus is to clarify the intended meanings of terms used in specific domains. The concrete representation of the language is captured in a metamodel [22].

The framework defines a number of properties that must be adhered to for an isomorphism to exist between the reference ontology and the metamodel. The underlying idea is that if such an isomorphism exists, it allows humans who interpret models in that language to do so clearly due to the precise correlation between the modeled artifacts and the domain concepts represented by
them. On the contrast, mismatches violating such an isomorphism introduce ambiguities or other forms of inclarities to the model.

### 6.1.1 On ontologies

Many slightly different definitions of the concept **ontology** can be found in literature. In this work we will use Guizzardi’s definition of ontology: “Ontology is a conceptual specification that describes knowledge about a domain in a manner that is independent of epistemic states and state of affairs.” This definition emphasizes that ontologies are universal models of domains or models of known knowledge in a domain.

Ontologies know four structural parts. **Classes** represent important concepts of the domain, which are often organized in a taxonomy where a generalization specification mechanism is used. **Relations** are used to represent associations between a domain’s concepts. Typical relations are has, is-a and consists of. Besides these taxonomical relations an ontology may contain binary relations where the first argument is the domain, and the second a range. Thus, relations may not just be between classes, but also between a class and a datatype. Individuals in an ontology are represented by **Instances**. Finally, **Formal axioms** express propositions about the classes, relations and instances in the ontology, that invariantly hold true. Axioms and instances need not necessarily be part of an ontology, but can be encoded as a data in a database (instances) or programmed piece of code of the system enforcing constraints (insuring axioms) where an ontology is used.

### 6.1.2 Mismatches

Guizzardi et al. specify in [13] four different kinds of mismatches that can be found comparing a reference ontology and a metamodel for a domain. These mismatches are precisely the observations that prevent a bijection to exist between ontology and metamodel. They are categorized as **lucidity**, **soundness**, **laconicity** and **completeness** as illustrated in Figure 6.1.

A model $M$ is said to be **lucid** with respect to a domain abstraction $A$ iff every construct in model $M$ represents at most one concept in the domain abstraction. Failing this we have a case of **construct overload**, i.e. a single language construct used to represent two or more domain concepts. Construct overload is an undesirable property as it causes ambiguity. When such a construct overload exists, additional knowledge needs to be stored alongside the concept being modeled in order to be able to understand this precisely what this concept represents. For example, when both Workers and IOWorkers are represented by a single concept ECU, then some additional attribute should be coupled with each ECU to store whether it represents a Worker or IOWorker.

**Soundness** is a property of a domain abstraction found when every construct in model $M$ represents at least one but perhaps multiple concepts in the domain abstraction $A$. Unsoundness leads to **construct excess**, a case where redundant language constructs represent no domain concept. Note that construct excess at the language level does not imply unsoundness at the modeling level; it is perfectly possible to define sound models despite the presence of construct excess in the language, by simply not using any unsound constructs.

A model $M$ is said to be **laconic** w.r.t. a domain abstraction $A$ when every abstraction concept is represented by at most one construct in the model representation $M$. This notion strongly relates to **construct redundancy** at the language level. Intuitively, this occurs when multiple language constructs can be used to represent one and the same domain concept.

The final type of mismatch is dubbed **completeness**. This property is given to models where every concept in the domain abstraction is represented by at least one construct in the repre-
Let's break down the text:

**CHAPTER 6. METRIC: METAMODEL APPROPRIATENESS**

(a) Lucidity:

(b) Soundness:

(c) Laconicity:

(d) Completeness:

Figure 6.1: Four different categories of mismatches. Lines represent a mapping between ontology and metamodel concepts, and dots represent domain concepts. Green dots map correctly in their category, while red dots indicate mappings with a violation in their category.

sentation. This is arguably the most important property that should hold for any representation system, as without it the domain contains inexpressible concepts.

6.2 Metric design

Inspired by this idea of identifying mismatches between a domain abstraction and accompanying language representation, we have defined a metric which has the purpose of giving insight in how well a metamodel relates to a given domain. In our context, we can consider the domain abstraction to be the best available metamodel for this domain. In case of the TWINSCAN domain this role can be fulfilled by the CARM 2G languages, and in case of the HSI domain this role can be fulfilled by the metamodel we developed and described in Chapter 5. For both domains we will relate these metamodels to the relevant parts of the PREEvision metamodel.

6.2.1 Approach

Compared to the framework by Guizzardi, we have a more explicit scenario where we compare a metamodel encoded using the Eclipse Modeling Foundation’s Ecore meta-metamodel with another Ecore compatible metamodel. Additionally, we wish to numerically quantify the suitability or fit of the target metamodel for the given domain. To this end, when comparing two metamodels $MM_1$ and $MM_2$, we will enumerate every concept in $MM_1$ and count the number of mismatches introduced when representing this concept or any of its associated relations in $MM_2$. In the following we make explicit how we enumerate metamodels and what categories of mismatches we consider, and how these differ from Guizzardi’s suggested mismatches.

Since we plan to evaluate our metric it is of importance that we have a well-defined way of
CHAPTER 6. METRIC: METAMODEL APPROPRIATENESS

enumerating all 'things' found in a domain. A particularly intuitive and simple way is to consider
the elements found in the representation of the metamodel describing the domain abstraction.
Primary candidates for such elements are the nodes found in an Ecore metamodel diagram. These
cover the ontology components analogous to classes and relations. This brings us to the following
list of items:

- **Class**: represented in the Ecore model as an EClass, used to depict some class in an object-
  oriented worldview.

- **Attribute**: represented in the Ecore metamodel as an EAttribute of an Eclass, used to
depict an attribute or property of objects.

- **Reference**: represented as lines in Ecore diagrams, specifying a relation between two classes
  of objects.

- **Data type**: represented as EDataType in Ecore, these can represent either enumerations or
  aliases to Java classes.

Classes fulfill a primary role in metamodeling as they mirror the entities found in the meta-
model's domain. Attributes embed the metadata of the entities, and references are used to repre-
sent relations between entities. Both attributes and references are a form of typed data. Enforcing
type validity at the metamodel level provides a strong mechanism for ensuring validity of models
themselves and relieves this task from user-code. As such, we place great value in the ability to
define custom datatypes at the metamodel level.

6.2.2 Additional mismatch categories

Inspired by the notion of mismatches by Guizzardi et al. we propose a set of categories to check
the metamodel concept against. We adopt the laconicity, completeness and lucifity mismatches,
but drop soundness. Furthermore we introduce three additional categories. We will first list the
resulting mismatch categories, and then argue why soundness is dropped.

- **Laconicity**: we will use the definition of laconicity precisely as given by Guizzardi et al., i.e.
  we have a mismatch when a single language construct is used to represent multiple domain
  concepts.

- **Completeness**: we will use the definition of completeness precisely as given by Guizzardi
  et al., i.e. we have a completeness mismatch when a concept or relation in $MM_1$ cannot be
  expressed in $MM_2$.

- **Lucidity**: we will use the definition of lucidity precisely as given by Guizzardi et al., i.e.
  we have a mismatch when a single metamodel construct is used to represent two or more
  domain concepts.

- **Type inaccuracy**: this occurs when typedness of some domain concept can be violated
due to the language construct chosen to represent this. Type inaccuracy mismatches are
found when a field with a too little restrictive format or validation is used to store data or
references. Primary example of such a type inaccuracy is the string-typed value attribute of
PREEvision Generic Attributes which are used to store numerous different kinds of data such
as integer or floating-point numbers, or even references. Strong-typing at the metamodeling
leveling alleviates the need for manual validation in model instances, which we consider
characterising for a metamodel that strongly fits (meta)data of entities. Failure to embed
or enforce type validation in some way thus leads to a worse fit of the target metamodel for
the domain under consideration.

- **Ontological vs. linguistic instantiation**: linguistically instantiated source objects should
  not be instantiated by user-defined definitions. An example of this would be representing
CARM 2G IOWorkers by instantiations of Logical Functions in PREEvision. The CARM 2G LogicalPlatform language contains a class concept for these IOWorkers. In our second alternative for modeling the LogicalPlatform in PREEvision, we would instantiate these IOWorkers from their Building Block Types which we defined in the PREEvision library ourselves. The motivation for considering this as a mismatch is twofold: first, typically a finer degree of flexibility is obtained by instantiating user-defined constructs as opposed to linguistically provided constructs. Second, use of ontological instantiation where this is not appropriate suggests the possibility of unintended reuse.

- **Intended meaning mismatch:** occurs when some concept is misemployed according to its intended meaning. A transformation between metamodels may decide to employ some linguistically provided concept for purposes other than intended by the target metamodel, which is likely to cause confusion to end-users of the metamodel at some point. For example, modeling a *sampling frequency* (of a CARM 2G servo group) in the property named *memory size* (of a PREEvision ECU) can be considered a mismatch in intended meaning despite their compatibility from a technology viewpoint. For our purposes, identifying this kind of mismatch requires human judgement.

We believe Guizzardi’s soundness mismatch is of minimal value. This would unnecessarily penalize any metamodel with unused features. Since these features are unused, there can not be any confusion interpreting such models. In determining the compatibility of a metamodel for some domain, additional features in the metamodel may even end up being useful later on.

6.2.3 Scale

Simply counting the mismatches incurred by evaluating a model transformation does not yield a metric with an outcome relative to size of the domain described by the tested metamodel. In order to obtain this desirable property, we also identify the number of concepts and mismatch tests for which no violations are found. This enables us to express the degree of metamodel compatibility as a value in range \([0, 1]\), without being influenced by a domain’s size.

6.3 Evaluation

Now that we have established a metric that gives us some indication of the fit between two metamodels, we are able to evaluate this metric on the domains and metamodels that we have studied during this project. In this section we present and discuss the outcome of our metric evaluation.

For our two case studies, the mismatches lucidity, soundness and completeness as suggested by Guizzardi et al. will not occur. Lucidities would occur when multiple source concepts or relations in \(MM_1\) are bundled in a single concept or relation in \(MM_2\). This is unintuitive and has simply not occurred in the transformations we developed. Soundness is not considered because the PREEvision metamodel is full of constructs that we make no use of, such as employee management or the built-in variance modeling framework. Deeming these features as soundness mismatches makes no sense. Completeness is actually a non-issue as, due to PREEvision’s Generic Attributes allowing arbitrary information to be encoded in the model, we know beforehand that the we will not encounter any inexpressibilities. Naturally, obtaining completeness in this fashion results in likely penalties due to laconicity, type- and intended-meaning mismatches. Because of the above, our evaluation excludes mismatches in these categories.

6.3.1 PREEvision appropriateness for TWINSCAN domain

We evaluated the metric as described in the previous section to obtain the fit of PREEvision as a tool for modeling the TWINSCAN domain. This was done by enumerating the concepts from the CARM 2G languages by going through the Ecore metamodels of these languages, while performing...
the metric inaccuracy checks against the PREEvision counterparts as obtained when applying the model transformation described in Chapter 4. The outcome of this metric evaluation is presented in the tables below.

For each of the tables below, analysed concepts are ordered by type as columns. The rows show with the first number in green the number of domain concepts according to its ontology that map to the analysed metamodel without introducing inaccuracies, while the number in red indicates how many concepts are found that would introduce inaccuracies. This means that for the PGAPP language, all of the 5 Class concepts introduce a laconicity inaccuracy.

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>0/5</td>
<td>0/12</td>
<td>2/2</td>
</tr>
<tr>
<td>Type</td>
<td>2/3</td>
<td>1/11</td>
<td>2/2</td>
</tr>
<tr>
<td>Instantiation</td>
<td>4/1</td>
<td>12/0</td>
<td>2/2</td>
</tr>
<tr>
<td>Meaning</td>
<td>5/0</td>
<td>12/0</td>
<td>4/0</td>
</tr>
</tbody>
</table>

Table 6.1: PGAPP language

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>3/2</td>
<td>4/2</td>
<td>1/0</td>
</tr>
<tr>
<td>Type</td>
<td>4/1</td>
<td>4/2</td>
<td>1/0</td>
</tr>
<tr>
<td>Instantiation</td>
<td>4/1</td>
<td>4/2</td>
<td>1/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>4/1</td>
<td>4/2</td>
<td>1/0</td>
</tr>
</tbody>
</table>

Table 6.2: PGSG language

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>3/11</td>
<td>0/4</td>
<td>0/0</td>
</tr>
<tr>
<td>Type</td>
<td>5/9</td>
<td>0/4</td>
<td>0/0</td>
</tr>
<tr>
<td>Instantiation</td>
<td>14/0</td>
<td>0/4</td>
<td>0/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>12/2</td>
<td>4/0</td>
<td>0/0</td>
</tr>
</tbody>
</table>

Table 6.3: PGWB language

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>4/2</td>
<td>3/1</td>
<td>5/0</td>
</tr>
<tr>
<td>Type</td>
<td>4/2</td>
<td>3/1</td>
<td>4/1</td>
</tr>
<tr>
<td>Instantiation</td>
<td>5/1</td>
<td>3/1</td>
<td>5/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>5/1</td>
<td>4/0</td>
<td>5/0</td>
</tr>
</tbody>
</table>

Table 6.4: LogicalPlatform language, alternative 1 (PV Hardware Architecture)

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>3/1</td>
<td>4/1</td>
<td>2/0</td>
</tr>
<tr>
<td>Type</td>
<td>3/1</td>
<td>4/1</td>
<td>2/0</td>
</tr>
<tr>
<td>Instantiation</td>
<td>4/0</td>
<td>5/0</td>
<td>2/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>4/0</td>
<td>5/0</td>
<td>2/0</td>
</tr>
</tbody>
</table>

Table 6.5: AppMap, alternative 1
CHAPTER 6. METRIC: METAMODEL APPROPRIATENESS

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>0/6</td>
<td>2/1</td>
<td>3/2</td>
</tr>
<tr>
<td>Type</td>
<td>4/2</td>
<td>2/1</td>
<td>5/0</td>
</tr>
<tr>
<td>Instantiation</td>
<td>4/2</td>
<td>2/1</td>
<td>5/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>3/3</td>
<td>3/0</td>
<td>4/1</td>
</tr>
</tbody>
</table>

Table 6.6: LogicalPlatform language, alternative 2 (PV Logical Function Architecture)

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>0/4</td>
<td>4/1</td>
<td>0/2</td>
</tr>
<tr>
<td>Type</td>
<td>3/1</td>
<td>4/1</td>
<td>2/0</td>
</tr>
<tr>
<td>Instantiation</td>
<td>4/0</td>
<td>5/0</td>
<td>2/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>1/3</td>
<td>5/0</td>
<td>0/2</td>
</tr>
</tbody>
</table>

Table 6.7: AppMap, alternative 2

The obtained results reveal that the primary difficulty lies with the mapping of attributes and ensuring their type validity. This is confirmed by our frequent need to employ Generic Attributes in combination with tedious consistency rules. The PGSG language maps rather well to PREEvision. Our case study agrees here, seeing as the PREEvision library nicely accommodates the define and instantiate paradigm found when instantiating servo groups from the PGSG language in an application described by a PGAPP model.

For the Logical Platform we evaluated the metric for both alternative architectures we considered. The metric suggests that staying true to the PREEvision philosophy of top-down refinement introduces fewer inaccuracies than when remaining closer to the CARM 2G Y-chart paradigm, i.e. where we encode the Logical Platform in PREEvision’s Logical Function Architecture. This matches with our intuition as using the Hardware Architecture and built-in mapping mechanism for the AppMap language yielded nearly no problems. Consistency rules nicely accommodate the built-in mapping mechanism to obtain type-safety for the AppMap language, whereas block ports where used in a confusing way in the alternative.

We see that the metric we defined generally agrees with the observations we made throughout the project. Since we have not modeled the Physical Platform in PREEvision, the metric fails to address some aspects. One result of the alternative of defining the Logical Platform in PREEvision’s Hardware Architecture, is the consequence for defining a mapping to an eventual Physical Platform, which would be virtually impossible since PREEvision offers no deeper refinement layers that would form a good fit around the Physical Platform. Had we forced an encoding of the Physical Platform in PREEvision, this would have great negative impact on the metric result for this architecture alternative, whereas the same set of ‘hacks’ for defining AppMap could be repeated to model the mapping between Logical and Physical Platform, when modeling the latter in the Logical Function Architecture. The metric does show that for each of the CARM 2G languages, defining custom attributes on domain entities without having a strong system to define references between entities forms one of the core problems of any PREEvision based approach.

6.3.2 Fit of PREEvision for HSI domain

We also evaluated the metric to obtain the fit of PREEvision as a tool for modeling the HSI domain. This was done by performing the metric inaccuracy checks against the concepts from the HSI metamodel as illustrated in Chapter 5. The outcome of this metric evaluation is presented in Tables 6.8-6.11. We did not consider entity attributes during this metric evaluation. We have already found that this is one of PREEvisions weak points during our feasibility studies and do not wish for this to have an overshadowing influence of the metric score overall. The importers
and exporters that deal with loosely structured data are vigorously able to handle this structure and this structure alone. The expected return on investment for properly structuring this data is considered minimal and as such, this was not performed in the EMF metamodel of the HSI. Furthermore, since datatypes in the EMF metamodel of the HSI are an exact clone of their PREEvision counterpart this was part not evaluated either, as the only possible outcome here is a perfect match. The results of evaluating the metric are given in the tables below.

<table>
<thead>
<tr>
<th>Class</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>7/4</td>
<td>10/6</td>
</tr>
<tr>
<td>Type</td>
<td>10/1</td>
<td>9/7</td>
</tr>
<tr>
<td>Instantiation</td>
<td>11/0</td>
<td>16/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>11/0</td>
<td>16/0</td>
</tr>
</tbody>
</table>

Table 6.8: Metric evaluation on HSI subsystem.

<table>
<thead>
<tr>
<th>Class</th>
<th>Reference</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>2/27</td>
<td>3/0</td>
</tr>
<tr>
<td>Type</td>
<td>2/27</td>
<td>3/0</td>
</tr>
<tr>
<td>Instantiation</td>
<td>11/18</td>
<td>3/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>29/0</td>
<td>3/0</td>
</tr>
</tbody>
</table>

Table 6.9: Metric evaluation on Class, Type, Instance paradigm.

<table>
<thead>
<tr>
<th>Class</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>0/4</td>
</tr>
<tr>
<td>Type</td>
<td>0/4</td>
</tr>
<tr>
<td>Instantiation</td>
<td>4/0</td>
</tr>
<tr>
<td>Meaning</td>
<td>4/0</td>
</tr>
</tbody>
</table>

Table 6.10: Metric evaluation on IOClusters part of HSI.

<table>
<thead>
<tr>
<th>Class</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laconicity</td>
<td>0/7</td>
</tr>
<tr>
<td>Type</td>
<td>7/0</td>
</tr>
<tr>
<td>Instantiation</td>
<td>5/2</td>
</tr>
<tr>
<td>Meaning</td>
<td>6/1</td>
</tr>
</tbody>
</table>

Table 6.11: Metric evaluation on ApplicationSpecificBoard part of HSI.

The metric reveals that the high-level structure of a HSI subsystem is rather well encoded in PREEvision. The high number of laconicity mismatches is purely due to the fact that each of the eight different class and type concepts of the ASML class/type/instance hierarchy are modeled as a separate class in our Ecore model, whereas they all map to Logical Function Types in PREEvision. This does not result in many typing inaccuracies due to the widespread use of classifier attributes and consistency rules though. Exploring the structure of HSI models deeper, we find that more and more laconicity and type inaccuracies are found. This follows from the fact that Logical Functions and Generic Attributes are used to encode large parts of the information in the HSI, whereas these concepts each receive at least their own EClass in the Ecore metamodel of the HSI.
Similar as for the PGSG and PGWB languages, the support for type instantiation is again rather well supported by PREEvision. This is partially inaccurate but obscured due to the fact that the mechanism used in the EMF HSI metamodel is not entirely true to its domain either. The one truly pragmatic way of achieving this requires the foundations of the used metamodeling technologies to start considering instantiation as a primary citizen in metamodeling, for example using approaches described by Atkinson et al. in [1].
Chapter 7

Conclusions and future work

The CARM 2G languages for ASML’s TWINSCAN domain were introduced with the intent of fulfilling two goals, namely 1) separating domain concepts from their implementation details, and 2) dividing different development disciplines by the Y diagram paradigm. We have studied these languages extensively and conducted a feasibility study of mapping the concepts used throughout the CARM 2G languages to an automotive electronics engineering architecture named EAST-ADL. We first reimplemented an available CARM 2G model of a temperature controller in the EAST-ADL implementation PREEvision. This revealed some of the shortcomings of both EAST-ADL and PREEvision, but also gave an intuition on how arbitrary CARM 2G models could be transformed into equivalent PREEvision ones. We continued by defining a textual transformation that accomplishes this. At all points we strived for a mapping that does not cause loss of expressivity while still making sense from an automotive point of view. Occasionally no obvious best way of mapping concepts is immediately clear. This resulted in our evaluation of two different architectures for the LogicalPlatform and AppMap languages, which both yield negative as well as positive results. Having completed this transformation we were able to formulate the outcome of the aforementioned feasibility study. It is our belief that PREEvision is not a suitable tool to assume the role of modeling the TWINSCAN domain. Although PREEvision can be used to represent any model due to its flexible Generic Attributes concept, this approach is filled with inconveniences. For example, adding arbitrary attributes to any concept is cumbersome and to achieve even a minimal sense of type correctness we need to resort to the tedious Consistency Framework. Additionally, the multi-layer architecture allows for separation of the Application, AppMap and LogicalPlatform, but the benefits of the Y diagram paradigm are lost. We asserted our impressions by introducing a metric, which confirms our intuition.

We also studied the domain of ASML’s Hardware Software Interface, which is modeled in the EAST-ADL based tool PREEvision. During this process the same difficulties we identified during transformation of the CARM 2G languages were encountered. In an ideal situation, the HSI would contain platform information only, and not be bloated with logic from the control application. This is not currently the case, and in order to remedy this the data stored in PREEvision HSI models needs to be positioned against their CARM 2G counterparts. A first step in this long-term process is the formalization of the HSI structure, which we made a start with during this project. EDEV encoded their proposed metamodel in PREEvision’s Logical Function Architecture while making extensive use of Generic Attributes to model large parts of information trees. We first established a technological bridge from PREEvision to the Eclipse Modeling Framework. To this end we extracted the relevant parts of the PREEvision metamodel and cloned these in an EMF Ecore metamodel, of which we then obtain instances from PREEvision models using a Java transformation embedded in a PREEvision metric snippet. Next we reimplemented EDEVs proposed metamodel as an Ecore model and defined a QVTo transformation between the extracted PREEvision models to instances of this new metamodel, thereby opening the possibility of positioning the HSI concepts to their respective CARM 2G languages.

Determining metamodel appropriateness for describing a domain
Based on a framework by Guizzardi et al. [13] we introduced a metric which quantifies how well a metamodel fits a given domain for which a reference metamodel exists. Guizzardi identifies a number of properties must be adhered to for an isomorphism to exist between a reference ontology and a metamodel of some domain. This has as underlying idea that if such an isomorphism exists, it allows humans who interpret models in that language to do so clearly due to the precise correlation between the modeled artifacts and the domain concepts represented by them. On the contrast, mismatches violating such an isomorphism introduce ambiguities or other forms of inclarities to the model. We introduced additional mismatch categories and established a method that enables quantification of the framework in the form of a metric. This metric represents how well the target metamodel fits the same domain by enumerating concepts from the reference metamodel, by identifying mismatches involved when mapping to the imperfect metamodel.

We then executed the evaluation of our metric on the TWINSCAN and HSI domains. Similar to our expectations, the metric reveals the weak points of PREEvision-based solutions. We have set the preliminaries for the positioning of the HSI concepts to the CARM 2G languages. This will reveal the concepts that erroneously are part of the HSI, i.e. those that do not map to the CARM 2G PhysicalPlatform. This eventually allows these concepts to by removed from the HSI, thereby improving the separation of implementation details. This activity is part of the 2014 roadmap.

The proposed metric is still open for improvements. Additional mismatch categories may be desirable. Currently, instances and formal axioms as part of an ontology are not taken into account since these are not part of the Ecore meta-metamodel and thus served no use in this work. It should be researched whether some automated evaluation of the laconicity, lucidity, soundness, completeness and typedness mismatches is possible when given a transformation from reference metamodel to the target metamodel. One approach for this would make use of QVTo traces found when executing an automated transformation. This does pose as additional requirement not only the two metamodels, but also a model instance of the reference metamodel and transformation between the two metamodels. Tuning mismatch weights of the metric could make the metric more expressive. Finally, the metric should be evaluated on a larger set of metamodel pairs to be able to come up with better verdicts regarding metamodel compatibility, as currently we have no statistical support to claim that, say, 0.8 is a good metric score.
Bibliography


Determining metamodel appropriateness for describing a domain 55


[19] Ramon Schiffelers, Wilbert Alberts, and Jeroen Voeten. Model-based specification, analysis and synthesis of servo controllers for lithoscanners. 8, 28

