View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by tuprints

Platform Independent Specification of Engineering Model Transformations

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von

Dipl.-Ing. Michael Schlereth

geboren am 22.04.1966 in Schweinfurt

Referent:	Prof. Dr. rer. nat. Andy Schürr
Koreferent:	Prof. Dr.-Ing. Ulrich Epple
Tag der Einreichung:	23.01.2014
Tag der mündlichen Prüfung:	13.06.2014

D17

Darmstadt 2014

I dedicate this thesis to Susan,
the sunshine of my life.

Acknowledgments

I thank my principal advisor, Prof. Dr. Andy Schürr, for giving me the opportunity to work as an external doctoral student at Real-Time Systems Lab in Darmstadt, for his open door if support was needed, and for sharing his deep insights into model transformation systems when it comes to the industrial application of theoretical concepts. My second advisor Prof. Dr. Ulrich Epple is one of the drivers of modeling and object-orientation in automation engineering. I thank him for inspiring discussions, for sharing ACPLT, and for supporting my thesis.

I thank my fellow doctoral students at Real-Time Systems Lab, Darmstadt, Anthony Anjorin, Marius Lauder, and Sebastian Rose, for sharing ideas and accepting me in the team. Tina Kraußer from Chair of Process Control Engineering, Aachen, helped me starting with the ACPLT process control system.

I thank my managers at Siemens AG, Ulrich Welz and Rolf Florschütz, for giving me the freedom to work on this thesis beside my daily work. At Siemens AG, I also thank my colleagues and friends Clemens Dinges and Rumwald Hermann for the work on exciting projects, for believing in me, and for motivating me to continue the work on this thesis.

Abstract

Production machine engineering involves multiple engineering disciplines defining together the configuration of each machine. Each of these disciplines provides an engineering model, which influences engineering models from other disciplines and is itself influenced by other engineering models. Therefore, building a valid configuration of a production machine requires the reconciliation of engineering models of all involved engineering disciplines.

Up to now, execution of model reconciliations by model transformation systems was mainly considered for desktop model transformation environments. The analysis of engineering processes and customer applications of production machines revealed that the industrial application of model transformations requires the execution of the same model transformation specification on different execution environments depending on the initiator of the model reconciliation. An electrical engineer runs the model transformation on his desktop between locally installed engineering applications for small organizations or on his field programming device for commissioning scenarios. For complex systems and bigger organizations, model transformations are executed on an enterprise product lifecycle management (PLM) server. A machine operator triggers model transformations on the real-time system of an automation controller for on-site reconfigurable machines, e.g. by physically connecting a modular device to a production machine.

To tackle this scenario, this thesis presents a new application of the model driven architecture (MDA), which transforms a platform independent model transformation specifications (PIM-MT) to platform specific model transformation specifications (PSM-MT) by higher order transformations (HOT). For industrial usage, both the platform independent transformation specification and the platform specific execution reuse proven existing technology which is tailored and extended where needed. This allows for the stepwise introduction of model transformation technology in existing engineering and technology environments based on a classification scheme which was developed as part of this thesis. For the PIM-MT specification, the strict handling of references between engineering model elements from current model transformation specifications, which does not fit well the requirements of engineering models with temporarily violate references within the engineering workflow, was replaced by a weaker reference handling based on domain specific reference designators. An existing model transformation specification, the ATL language, has been tailored for PIM-MT specifications. For the PSM-MT desktop execution, the ATL desktop model transformation engine was reused. XSL transformations were adapted for enterprise model transformations executed on PLM servers. A PSM-MT engine for real-time IEC 61131 programmable logic controllers was developed as part of this thesis.

With the results of this thesis it is now possible to build a customized engineering environment as an extension of the existing infrastructure of a machine builder, which automates the configuration of production machines by using model transformations generated from a common platform independent specification on multiple execution platforms.

Zusammenfassung

An der Entwicklung von Produktionsmaschinen sind mehrere Entwicklungs-Fachrichtungen beteiligt, insbesondere Mechanikkonstruktion, Elektrokonstruktion und Automatisierungsentwicklung, die zusammen die Konfiguration einer Maschine erstellen. Jede dieser Fachrichtungen beschreibt die Maschinenkonfiguration mit Hilfe eines fachspezifischen Maschinenmodells, das einerseits die Inhalte anderer fachspezifischer Maschinenmodelle beeinflusst und andererseits selbst von den Inhalten anderer fachspezifischer Maschinenmodelle beeinflusst wird. Daher müssen bei der Entwicklung einer Produktionsmaschine die Maschinenmodelle aller beteiligten Fachrichtungen untereinander abgeglichen werden.

Der Abgleich von Maschinenmodellen durch Modelltransformationssysteme wird bisher hauptsächlich auf Arbeitsplatzsystemen mit lokal installierten Modelltransformations-umgebungen durchgeführt. Die Analyse der Entwicklungsprozesse und Kundenanwendungen von Produktionsmaschinen zeigt aber, dass die industrielle Anwendung von Modelltransformation erfordert, dass die gleiche Modell-transformationsspezifikation je nach Initiator des Modellabgleichs auf verschiedenen Ausführungsumgebungen durchgeführt werden muss. In einer kleinen Firma oder bei einer Maschineninbetriebnahme führt ein Elektrokonstrukteur die Modelltransformation auf seinem Arbeitsplatzrechner oder Programmiergerät zwischen lokal installierten Entwicklungssystemen aus. Für komplexe Maschinen oder größere Entwicklungsorganisationen werden Modelltransformationen regelmäßig auf einem Enterprise Product Lifecycle Management (PLM) Server ausgeführt. Ein Maschinenbediener startet Modelltransformationen auf dem Echtzeit-System einer speicherprogrammierbaren Steuerung (SPS) für Maschinen, die Vor-Ort rekonfiguriert werden können, beispielsweise durch den physikalischen Anschluss eines neuen Maschinenmoduls an eine Produktionsmaschine.

Um diese Aufgabenstellung zu bewältigen, präsentiert die vorliegende Arbeit eine neue Anwendung der Model-Driven-Architecture (MDA), bei der eine plattformunabhängige Modelltransformationsspezifikation (PIM-MT) in plattformspezifische Modell-transformationsspezifikationen (PSM-MT) mit Hilfe von Higher-Order-Transformations (HOT) transformiert wird. Um den industriellen Einsatz zu ermöglichen, wird sowohl für die plattformunabhängige Modelltransformationsspezifikation als auch für die plattformspezifische Modelltransformationsmaschinen auf erprobte existierende Technologien aufgebaut, die im Rahmen dieser Arbeit angepasst und erweitert wurden. Dadurch kann die Technologie von Modelltransformationen mit Hilfe eines Klassifikationsschemas, das im Rahmen der vorliegenden Arbeit erstellt wurde, schrittweise in vorhandene Entwicklungs- und Technologieumgebungen eingeführt werden.

Für die plattformunabhängige Modelltransformationsspezifikation (PIM-MT) wurde die strenge Definition von Referenzen zwischen Elementen des Maschinenmodells, die schlecht zu den Anforderungen von Maschinenmodellen mit zeitweise ungültigen Referenzen während des Entwicklungsprozesses passt, durch eine lockerere Definition von Referenzen mit Hilfe von domänenspezifischen Kennzeichnungssystemen ersetzt. Eine existierende Sprache zur Spezifikation von Modelltransformationen, die ATL Modelltransformationssprache, wurde zur Nutzung als PIM-MT angepasst. Zur PSM-MT-Ausführung auf Arbeitsplatzrechnern wurde die ATL-Modelltransformations-maschine wiederverwendet. XSL-Transformationen wurden für serverbasierte Modelltransformationen auf PLM-Systemen angepasst. Im Rahmen der vorliegenden Arbeit wurde eine neue Modelltransformationsmaschine entwickelt, die auf echtzeitfähigen speicherprogrammierbaren Steuerungen, die dem IEC 61131-3 Standard entsprechen, ausführbar ist.

Auf Basis der Ergebnisse der vorliegenden Arbeit ist es nun möglich eine kundenspezifisch angepasste Entwicklungsumgebung für Produktionsmaschinen zu erstellen, die die existierende Entwicklungsumgebung eines Maschinenbauers so erweitert, dass Produktionsmaschinen mit Hilfe von Modelltransformationen automatisiert auf verschiedenen Plattformen konfiguriert werden können. Die Spezifikation der plattformspezifischen Modelltransformationen wird dabei aus einer einmalig erstellten plattformunabhängigen Modelltransformationsspezifikation für alle Zielplattformen generiert.

Notations Used Within Figures

The architectural and implementation diagrams presented within this thesis use the UML notation [IS12a].

The more architectural related figures are created with Microsoft PowerPoint[1] (e.g. Figure 21). The architectural diagrams are class diagrams annotated with some graphic markers as shown with some samples in Figure 1. Classes are depicted by rectangles, the can symbol is used for classes stereotyped as models. Directed or undirected associations, dependencies, and generalizations are used according to the UML specification.

The graphic markers are mainly used for the grouping of elements. Elements with the same fill color belong to a group. Beside fill colors, boundary rectangles are used to group elements.

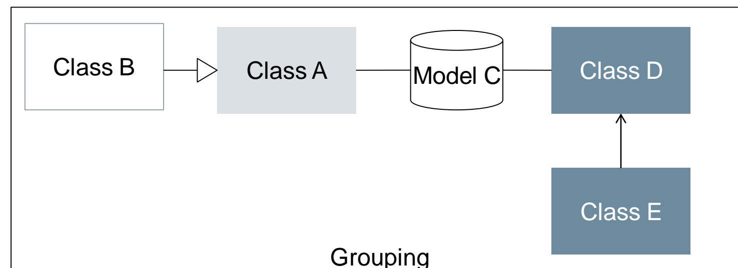

Figure 1: Architectural UML Diagram Sample

The more implementation related UML figures are created with the UML Modeling Tool "Enterprise Architect"[2] (e.g. Figure 11). The UML elements used are the same as for the more architecture related figures but the visual appearance is different.

Beside the formal UML figures, informal figures appear within this thesis where diagrams are cited from references (e.g. Figure 2), where diagrams shall visualize informal descriptions (e.g. Figure 7), or where figures show screenshots from implementation tools (like Ecore diagrams in Figure 77).

---

[1] https://office.microsoft.com/en-us/powerpoint/
[2] http://www.sparxsystems.com/products/ea/

Contents

1 Introduction 21
1.1 Motivation 21
1.2 Scope 23
1.3 Contributions 25
2 Application Scenario 27
2.1 Labeling Machine 27
2.2 Engineering Models 28
2.3 Engineering Model Reconciliation 30
3 Requirements for Engineering Model Transformations 33
3.1 Mission Statement 34
3.2 Model Driven Specification of Model Transformations 35
3.3 Requirements for Engineering Model Transformation Specifications 37
3.4 Structure of Engineering Model Transformation Specifications 39
3.5 Engineering Model Access 40
3.6 Related Work 44
4 Platform Specific Model Transformation Languages and Engines 45
4.1 Desktop Model Transformation Engine 49
4.1.1 Rule Language 50
4.1.2 System Model 51
4.1.3 Pattern Language 52
4.1.4 Inter-Rule Execution Control 55
4.1.5 Modularization 56
4.1.6 Implementation Alternatives 57
4.1.7 Summary 59

4.2 Enterprise Model Transformation Engine 60
4.2.1 Rule Language 62
4.2.2 System Model 64
4.2.3 Pattern Language.. 66
4.2.4 Inter-Rule Execution Control 69
4.2.5 Modularization 69
4.2.6 Implementation Alternatives 70
4.2.7 Summary... 71
4.3 Real-Time Model Transformation Engine... 73
4.3.1 Rule Language 77
4.3.2 System Model 78
4.3.3 Pattern Language.. 79
4.3.4 Inter-Rule Execution Control 82
4.3.5 Modularization 83
4.3.6 Related Work 84
4.3.7 Summary... 84
4.4 PSM-MT summary 86
5 Platform Independent Model Transformation Language 88
5.1 Rule Language.. 91
5.2 System Model 93
5.3 Pattern Language 96
5.4 Inter-Rule Execution Control.. 100
5.5 Modularization.. 103
5.6 Related Work... 104
5.7 Summary 105

6 PIM-MT to PSM-MT Transformations ... 107
6.1 PIM-MT transformation specification example ... 110
6.2 Desktop Model Transformations - ATL to ATL ... 114
6.2.1 PSM-MM to PIM-MM Transformation ... 115
6.2.2 Model Instances ... 116
6.2.3 PIM-MT to PSM-MT Transformation ... 117
6.2.4 Rule Execution ... 118
6.2.5 Summary ... 118
6.3 Enterprise Model Transformations - ATL to XSLT ... 119
6.3.1 PSM-MM to PIM-MM Transformation ... 120
6.3.2 Model Instances ... 122
6.3.3 PIM-MT to PSM-MT Transformation ... 124
6.3.4 Rule Execution ... 126
6.3.5 Summary ... 127
6.4 Real-Time Model Transformations - ATL to IEC 61131-3 ... 128
6.4.1 PSM-MM to PIM-MM Transformation ... 130
6.4.2 Model Instances ... 133
6.4.3 PIM-MT to PSM-MT Transformation ... 134
6.4.4 Rule Execution ... 137
6.4.5 Summary ... 139
7 Conclusion and Future Work ... 140
8 References ... 143
9 Appendix ... 148
9.1 PIM-MT to intermediate ATL transformation ... 148
9.2 Desktop PIM-MT/Intermediate Representation to PSM-MT transformation ... 153

9.3 Enterprise PIM-MT/Intermediate Representation to PSM-MT transformation .. 155

9.4 Real-Time PSM-MM to PIM-MM transformation 158

9.5 Real-Time PIM-MM generalizations transformation 160

9.6 Real-Time PIM-MT/Intermediate Representation to PSM-MT transformation.. 163

List of Figures

Figure 1: Architectural UML Diagram Sample 11
Figure 2: Mechatronics – synergy from the interaction of different disciplines [VD04] 21
Figure 3: Reconciliation of Mechatronic Models by Model Transformations 23
Figure 4: Using Different Technical Platforms for the Definition and the Execution of Model Transformations 24
Figure 5: Bottle Labeling Machine 27
Figure 6: Water Bottles with Different Labels Applied 28
Figure 7: Bottle Labeling Machine – Functional Structure 29
Figure 8: Bottle Labeling Machine – Electrical Devices 29
Figure 9: Bottle Labeling Machine - Software Structure 30
Figure 10: Reconciliation Between Electrical and Automation Engineering Models 31
Figure 11: Electrical and Automation Engineering Models 32
Figure 12: Model Reconciliation of Production Machines 33
Figure 13: Platform Independent Specification of Model Transformation Rules 35
Figure 14: The MDA model transformation pattern [MM03] 36
Figure 15: Actors and Use-Cases of Engineering Model Transformations 37
Figure 16: Model Transformation Specification Structure 40
Figure 17: Model Access Operations 41
Figure 18: Engineering Metamodel 41
Figure 19: Transformation Engine - Model Adaptation 43
Figure 20: Model Transformation Platform 45
Figure 21: Desktop Model Transformations 49
Figure 22: ATL Model Transformation Specification 50
Figure 23: The Ecore kernel [St09] 51
Figure 24: ATL model queries: InPattern 52
Figure 25: ATL model modifications: OutPattern 53
Figure 26: ATL rule example with InPattern and OutPattern 54
Figure 27: ATL rule execution order 55
Figure 28: ATL launch configuration 56
Figure 29: ATL modularization by superimposed modules 57
Figure 30: ATL PSM-MT features 59
Figure 31: PLM Environment of a Machine Builder [ES09] 60
Figure 32: Model Transformations between Authoring Systems [ES09] 61
Figure 33: Enterprise Model Transformations 61
Figure 34: Enterprise Data Transformation Technologies 62
Figure 35: XSL Transformations (XSLT) 62
Figure 36: Template Rule 63
Figure 37: PLMXML schema extension 65
Figure 38: Template Rule Patterns 67
Figure 39: XSLT template rule example with match pattern and sequence constructor 68
Figure 40: XSLT PSM-MT features 72

Figure 41: Real-Time Model Transformations 73
Figure 42: Basic functional structure of a PLC-system [In03a] 74
Figure 43: PLC Application Program 74
Figure 44: Program Organization Units (POU) 75
Figure 45: Data Type 75
Figure 46: PLC model example: Labeling Device 76
Figure 47: Transformation Rule Function Blocks 77
Figure 48: Building the Introspection Model 79
Figure 49: Meta Information about IEC 61131-3 Elements 80
Figure 50: Transformation Function Block Example 81
Figure 51: Rule Execution Program Example 82
Figure 52: IEC 61131-3 PSM-MT features 85
Figure 53: Model Driven Architecture PIM to PSM transformation 88
Figure 54: PIM-MT/PSM-MT Platform Scope 89
Figure 55: Original ATL Matched Rule Syntax 91
Figure 56: Generalized ATL Matched Rule Syntax 92
Figure 57: Platform Independent System Model 93
Figure 58: PSM-MM to PIM-MM Transformation 94
Figure 59: Pattern language with Expressions vs. User Defined Functions 97
Figure 60: Object Reference Handling by ATL Traceability Links 98
Figure 61: Weak Object References by Reference Designators 100
Figure 62: Platform Specific Execution Control 101
Figure 63: PIM-MT to PSM-MT Transformations 106
Figure 64: Implementation of the Higher Order Transformation (HOT) 107
Figure 65: PIM-MT Intermediate Representation 109
Figure 66: Implementation of the Metamodel Transformations 110
Figure 67: Platform Independent Electrical Metamodel (Ecore) 111
Figure 68: Platform Independent Automation Metamodel (Ecore) 112
Figure 69: Platform Independent Model Transformation (PIM-MT) Example 113
Figure 70: Desktop Model Transformation Evaluation 114
Figure 71: Electrical Model Example 116
Figure 72: Automation Model Example 117
Figure 73: Desktop PIM-MT to PSM-MT Higher Order Transformation (HOT) 118
Figure 74: Siemens Teamcenter 4-tier Architecture 119
Figure 75: Enterprise Model Transformation Evaluation 120
Figure 76: XML Schema Platform Specific Metamodel – Electrical Engineering Metamodel 121
Figure 77: Ecore Platform Specific Metamodel – Electrical Engineering Metamodel 122
Figure 78: XML Platform Specific Model Instance 123
Figure 79: XSLT Specific Model Transformation Rule 125
Figure 80: XSLT Rule Execution – Initial Template 126
Figure 81: Enterprise PIM-MT to PSM-MT Higher Order Transformation (HOT) 127
Figure 82: Platform Specific Model Transformation Language – IEC 61131-3 Real-Time Systems 128
Figure 83: Real-Time Model Transformation Evaluation Setup 129

Figure 84: ATL Rule to IEC 61131-3 Structured Text - Model Transformations Library 130
Figure 85: IEC 61131-3 Platform Specific Metamodel 131
Figure 86: IEC 61131-3 PSM-MM to PIM-MM Transformation 132
Figure 87: IEC 61131-3 Platform Specific Model Instances 134
Figure 88: IEC 61131-3 Platform Specific Model Transformation Rule 136
Figure 89: IEC 61131-3 Platform Specific Rule Execution 138
Figure 90: Real-Time PIM-MT to PSM-MT Higher Order Transformation (HOT) 139

1 Introduction

The introduction starts with an overview about the requirements of automation systems for production machines, which are the motivation for the work presented in this thesis. The second section sets the scope of the work in relation to requirements of production machine builders and in relation to model transformation technologies. Finally, the summary of the contributions of the work presented closes the introduction.

1.1 Motivation

Production machines are complex mechatronic systems, built from mechanical elements, electrical elements and software elements. These different disciplines, mechanics, electrics, and software, must be reconciled within engineering of a production machine as well as part of the operation of a production machine. Therefore, models, which describe the structure of elements in each discipline, can be used for machine engineering as well as for machine operation.

Most contributions about reconciliation of these discipline models, for example the VDI 2206 guideline [VD04] shown in Figure 2, focus on the engineering of a production machine and not on the operation of production machines. Dependent on the organization of a machine builder, reconciliation of the discipline specific engineering models might be executed on a desktop level (on a personal engineering PC) or on an enterprise level (e.g. as part of a server based product lifecycle management (PLM) system [ES09]).

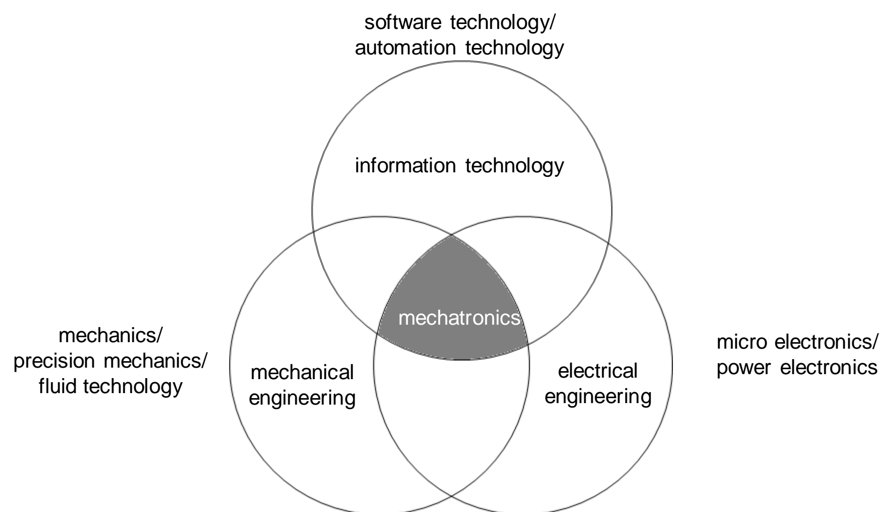

Figure 2: Mechatronics – synergy from the interaction of different disciplines [VD04]

The work presented in this thesis extends this engineering centric view of model reconciliation from the application of model reconciliation rules to the operation phase of production machines. Flexible manufacturing environments require reconfigurable machines at the shop floor instead of preconfigured machine configurations delivered by the machine builder. This flexibility can be provided depending on the machine users' requirements by the same set of rules, which can be executed either as part of an engineering environment or as part of the real-time automation system.

The reuse of model transformation rules in different execution environments supports different usage scenarios:

- An automation engineering system provider can provide a set of engineering rules for his automation system, which can be executed in different engineering environments depending on the existing infrastructure of a machine builder.
- The organization of a machine builder can grow from a desktop level, with engineering rules executed on a personal PC, to an enterprise level, with engineering rules executed on a PLM server system, without reimplementation of the existing engineering knowledge.
- A machine builder can either apply machine configuration rules in his engineering environment or deliver the same set of rules to a machine user for on-site reconfiguration of the machine as part of the automation system runtime.
- A machine user can use the machine models together with the discipline specific models for extension and adaptation to new manufacturing needs.

These requirements are not met by existing model transformation languages, which are restricted to a specific execution environment. Therefore, this thesis presents a new approach, which separates the model transformation language from the execution engine and which defines a transformation from platform independent model transformation languages to different platform specific model transformation execution environments.

1.2 Scope

The terms “model” and “model transformation” are used in the sense of the model driven architecture (MDA) [MM03], where models are considered as an object-oriented representation of data-structures and for example not in the sense of models used for mechanical structure analyses like finite elements models [Mi06]. Models are used to describe the discipline specific information like the mechanical model, which describes the mechanical structure of an automation system, or the electrical model, which describes the hardware and cabling of electrical devices (see Figure 3). These models are reconciled on changes by model transformations.

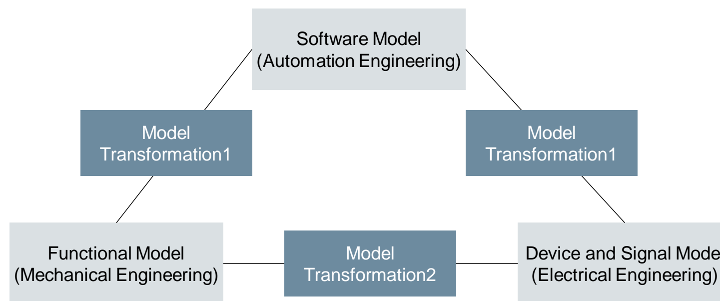

Figure 3: Reconciliation of Mechatronic Models by Model Transformations

To fulfill the requirement of our usage scenario about execution of model transformations in different execution environments, the definition of model transformations and the execution of model transformations can be based on different technical platforms. According to the MDA Guide [MM03], "A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented." The application considered within this thesis is an environment for model transformations, e.g. on a desktop level as part of an Eclipse workspace[3] or as part of a real-time automation system. This thesis presents a new approach, which allows for the mapping from the technical platform of the definition of a model transformation to the technical platform of the execution of a model transformation. The primary platform used for the definition of model transformations is the Java-based Eclipse platform together with its integrated modeling frameworks [St09].

[3] http://eclipse.org/

The focus of this thesis is on the reuse of the existing platform specific model transformation engines. Nevertheless, the platform specific model transformation environment has been extended where required. Especially for the execution of model transformation at real-time automation systems, the IEC 61131-3 [In03b] execution environment of programmable logic controllers has been extended to support the platform specific execution of model transformation specifications. These extensions were implemented on the application level of the model transformation environments and are compatible to the existing environment.

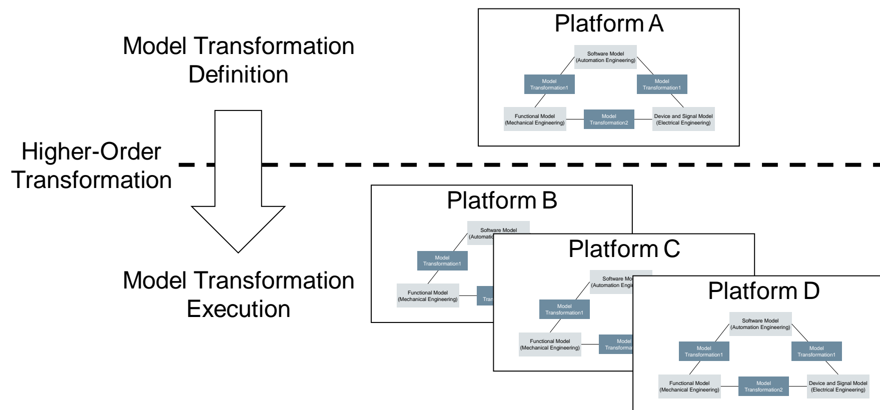

Figure 4: Using Different Technical Platforms for the Definition and the Execution of Model Transformations

1.3 Contributions

The starting point of this thesis was the observation that the theoretical concepts developed by the vivid model transformation community currently have a limited application in industrial environments. Therefore, the work presented in this thesis shows a new approach which allows for the application of new model transformation languages as part of existing industrial environment.

This thesis provides new results in the following areas:

- A new approach for platform independent model transformation authoring reusing existing model transformation languages was introduced. This approach was evaluated by the reuse of the ATL model transformation language [Jo06].
- A general model of the structure of model transformation languages was introduced. This structure consists of five elements, rule language, system model, pattern language, inter-rule execution control, and modularization, which together define the features of a model transformation language. An analysis according to this model is provided for the ATL desktop model transformation platform [Jo06], for the XSLT server based model transformation platform [Wo07], and for the IEC 61131 real-time model transformation platform [In03b].
- Based on this general model of the structure of model transformation languages, the existing concept of PIM to PSM transformations was extended to the transformation of platform independent model transformation languages (PIM-MT) to platform specific model transformation languages (PSM-MT). This PIM-MT to PSM-MT transformation was implemented for the transformation to the ATL desktop model transformation language [Jo06], for the server based XSL transformations [Wo07], and for real-time transformations implemented by the IEC 61131-3 structured text (ST) programming language.
- For the execution of model transformation on real-time programmable logic controllers (PLC), a model transformation engine was developed as part of this thesis by the structured text (ST) programming language. This model transformation engine can be executed as part of the automation program of production
- The analysis of the PIM-MT to PSM-MT transformation revealed that an opposite PSM to PIM transformation is required to provide the metamodels used by the platform independent model transformation authoring. Therefore, a specification model was developed, which allows for the handling of the reverse-directed model transformation required for the PIM-MT to PSM-MT as a single unit. This reverse transformation was implemented for the server based metamodel provided by XML schema definitions [Wo12] and for the IEC 61131-3 metamodel used by real-time model transformations [In03b].

Based on the main results, it is now possible to generate platform specific model transformation specification based on the requirements of machine builders for different customer orders and different development processes.

The contributions provided by this thesis are presented with the help of an application scenario based on the engineering models of a bottle labeling machine presented in Section 2.

The requirements for the new concept of model driven implementation of model transformations are introduced in Section 3. This concept is based on the higher order transformations from a platform independent model transformation specification (PIM-MT) to multiple platform specific model transformation specifications (PSM-MT) as a new application of the model driven architecture (MDA) [MM03]. Section 3 describes the structure of engineering model specifications, the structure of engineering models, and the interfaces to these engineering models. This given environment must be handled by the MDA approach for model transformation specifications introduced by this thesis.

The platform specific model transformations engines (PSM-MT) required by the application example of Section 2 are covered in Section 4. This section starts with the introduction of an analysis scheme for model transformation platforms, which was developed as part of this thesis. This analysis scheme is applied to the three model transformation platforms required within production machine engineering: desktop model transformations, server based model transformations, and real-time model transformations. The existing technical platforms for server based model transformations and for real-time model transformations could not be used as currently available, but had to be adapted and extended as part of this thesis. Section 4 shows the adaptation of XSL transformations and presents the IEC 61131-3 model transformation engine developed as part of this thesis.

The counterpart to the platform specific model transformations, the platform independent model transformation specification (PIM-MT), is subject of Section 5. The ATL model transformation language is adapted for the usage as a PIM-MT by selecting a subset of its language feature set and providing usage guidelines for its pattern language.

The implementation details of the PIM-MT to PSM-MT higher order transformation and the PSM-MM to PIM-MM transformation are described in Section 6. The implementation covers the transformation of ATL to ATL, ATL to XSLT, and ATL to IEC 61131-3.

Section 7 summarizes the results of this thesis and gives an outlook for future work.

2 Application Scenario

As outlined in Section 1.2 this thesis targets at the reconciliation of models as part of machine engineering and machine operation. As a running example, the engineering model of a production machine with optional modules will be used.

2.1 Labeling Machine

The production machine market is very broad and covers many industries like packaging, textile, plastics, handling, converting, and printing. The production machine used as an example comes from the packaging industry and is a labeling machine used as part of a bottle filling line (see Figure 5). In this application, the bottles filled with a beverage enter the labeling machine on an infeed conveyor belt as shown at the left of Figure 5. Then, the bottles are moved around a big wheel to make way for multiple labeling units. These labeling units apply labels to different areas of a bottle, e.g. to the front side of the bottle, to the back side of the bottle or to the neck of a bottle (see Figure 6).

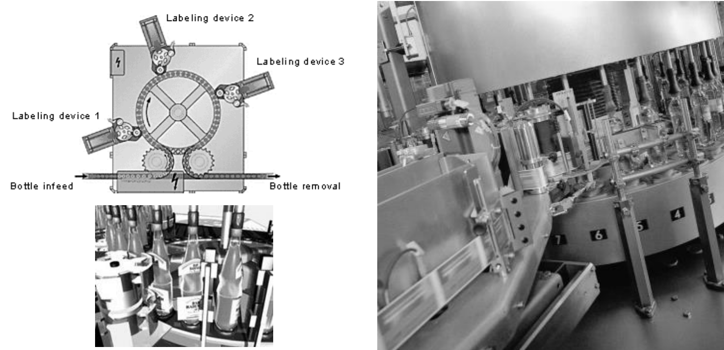

Figure 5: Bottle Labeling Machine

The configuration of the labeling units, which are part of the labeling machine, depends on the design of bottles and on the design of the label to be applied. The configuration of the labeling units includes the positioning of the label, the labeling technology used (e.g. hot or cold glue, self-adhesive), and the size of the label. Depending on the product, the configuration may be fixed for a long time if the bottles design remains unchanged or may be changed frequently, for example for marketing campaigns with specific designs or if the filling line is used in contracting for products of different customers.

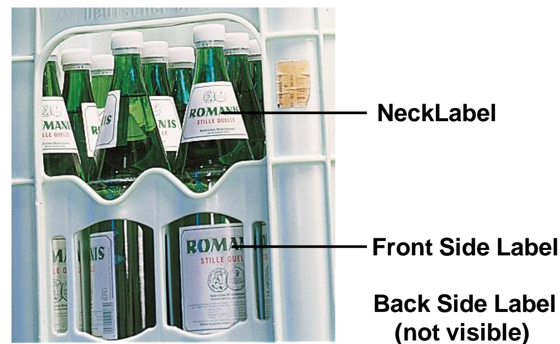

Figure 6: Water Bottles with Different Labels Applied

2.2 Engineering Models

The engineering process of a production machine like the labeling machine used in our example involves the collaboration of three engineering disciplines as presented in the introduction (see Section 1.1):

- Mechanical engineering defines the physical shape of the machine as a 3D model and the functional structure of the machine.
- Electrical engineering selects and configures the electrical devices such as programmable logic controller (PLC), electrical drives, and I/O devices.
- Software engineering develops the PLC application code.

The functional structure of the labeling machine is shown as a 2D sketch in Figure 7. The machine consists of a base unit with connectable labeling units. The base unit includes a switching cabinet with a master controller, while the labeling units include decentralized devices like the drives and IO required by the labeling unit.

For simplicity, in the application example only the reconciliation between information from electrical engineering and from software engineering will be presented. The introduced concepts can also be applied to the exchange between the other engineering disciplines.

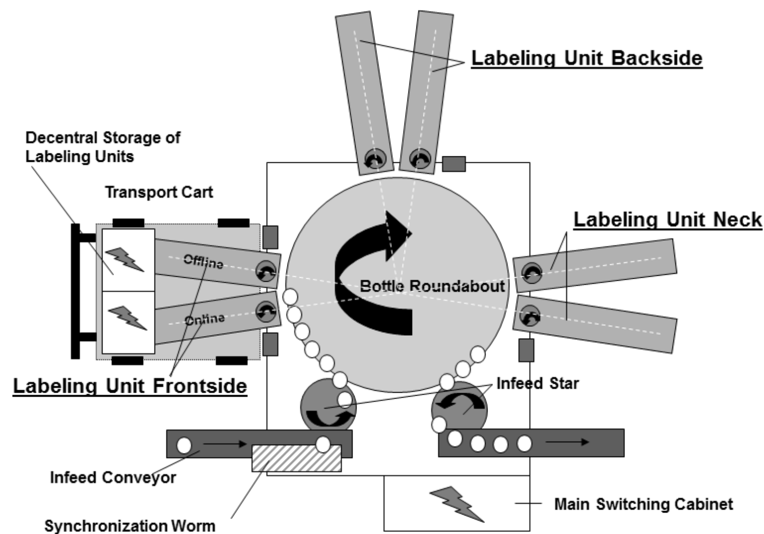

Figure 7: Bottle Labeling Machine – Functional Structure

The electrical device structure is shown in Figure 8. The main PLC is connected to the optional decentralized labeling units by a field bus network. If a labeling unit with its electrical devices is connected to or disconnected from the main unit, the main PLC must reconfigure the machine control software to adapt the labeling process to the changed configuration of the labeling machine.

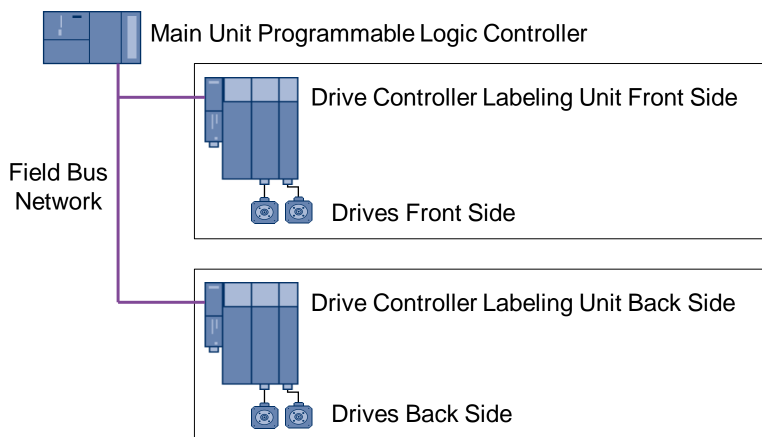

Figure 8: Bottle Labeling Machine – Electrical Devices

The automation software structure reflects the structure of the electrical configuration. As shown in Figure 9, the PLC software consists of an invariant part, which includes for example the task system and general execution logic. For each labeling unit, a control module for the labeling unit must be activated if the labeling unit is present. For addressing the electrical devices of the control unit from the labeling unit control module, interface modules like the axis technology object shown in Figure 9 must be activated. These interface modules address the drive controller by the field bus network, which connects the main PLC with the labeling unit stations.

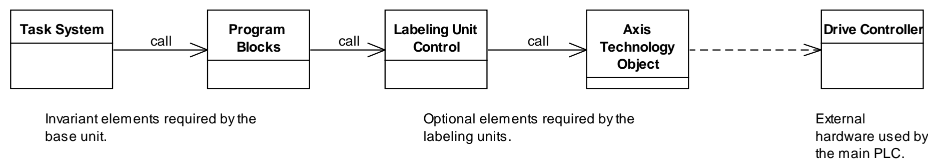

Figure 9: Bottle Labeling Machine - Software Structure

2.3 Engineering Model Reconciliation

For a valid configuration of a machine, the models of the different disciplines must be reconciled. For the application scenario considered here, changes in the electrical model must be reflected within the software model. For example, on a real-time controller for each labeling device drive controller within the electrical model, the corresponding software blocks in the automation model must be activated (see Figure 10). The abstract definition of the model reconciliation as shown in Figure 10 does not include the execution of the model transformation between the models. Depending on the system characteristics, the reconciliation can be executed by the construction of a complete model, by the addition of elements to an existing model, or by activating already existing elements within an existing model.

The reconciliation of engineering models must only handle model elements which are relevant for the model transformation specification. For example, the elements "FrontSide:LabelFeeder" and "BackSide:LabelFeeder" shown at the lower right in Figure 10 must be activated as part of the model reconciliation, but any relationship of these model elements to other elements in the automation software needs not to be handled by the reconciliation, because these relationships are not dependent on the source model but can be constructed within the target element as part of the target element initialization process without any knowledge from the source model. Therefore, these elements are modeled without any relationship to other model elements.

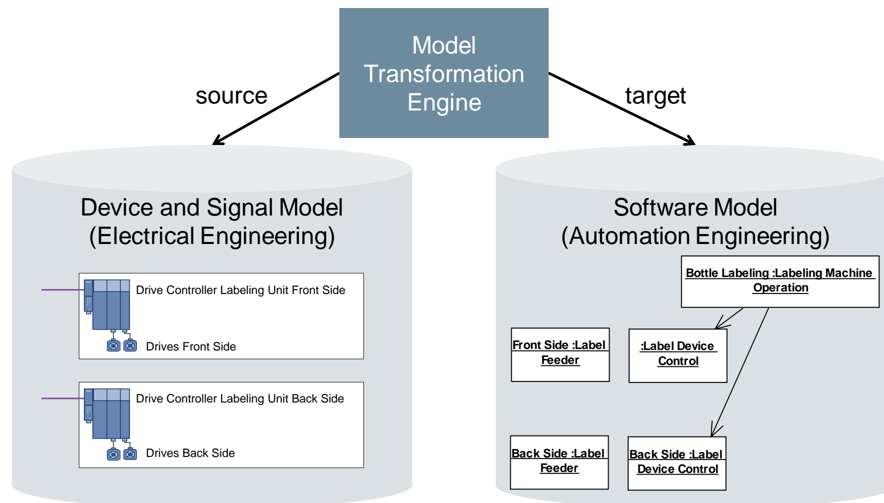

Figure 10: Reconciliation Between Electrical and Automation Engineering Models

The application example shown in Figure 10 is used throughout this thesis to evaluate the platform independent specification of model transformation systems. Within the evaluation, the model elements of the application example are used with the formal names shown in Figure 11. All elements include a name attribute, which provides a unique identifier for each element. This name is also used for references between elements.

The device and signal model is an instance of an electrical model. The electrical model consists of a DriveControlUnit, which represents the DriveController from the device and signal model shown in Figure 10. The DriveControlUnit models a vendor independent device. Within the application example, low voltage converters of the SIEMENS SINAMICS series[4] were used as vendor specific devices, namely the G120ControlUnit and the S120ControlUnit. The second part of the device and signal model, the drives used for front side and back side labeling, are represented by the DriveObject of the electrical model. The DriveObject includes an address attribute, which holds the communication address used on the field bus network to control the DriveObject. The DriveObject elements included in a DriveControlUnit are referenced by the driveObject attribute.

The software model is an instance of the automation model. It consists of a LabelDeviceCuControl, which is used in the user program to control the DriveControlUnit from the electrical model. The DriveObject from the electrical model is controlled by two different elements of the automation model. The DriveAxis specialization of a TechnologyObject is used to provide positioning control functions. The labeling function of the DriveAxis is controlled by the LabelDeviceControl element.

---

[4] http://www.industry.siemens.com/drives/global/en/converter/low-voltage-drives/Pages/Default.aspx

The labelDevice attribute of the LabelDeviceCuControl references the related LabelDeviceControl. The TechnologyObject and the LabelDeviceControl include an address attribute, which holds the communication address used on the field bus network of the related elements from the electrical model.

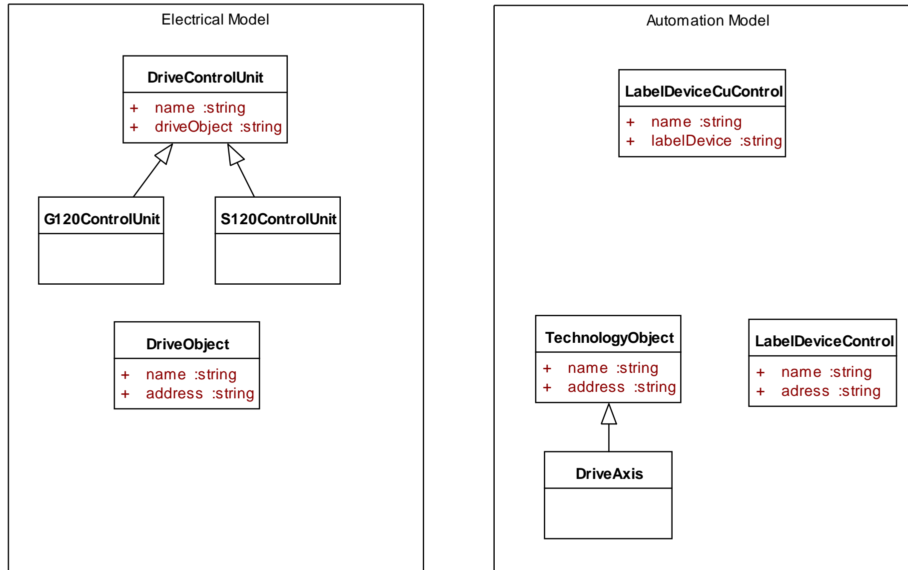

Figure 11: Electrical and Automation Engineering Models

The engineering models described in Section 2.2 , which are part of such a reconciliation operation, reside on different execution platforms depending on the reconciliation scenario, e.g. on desktop engineering applications, on a PLM server, or on automation controllers. The reconciliation platform required for a specific application depends on the engineering process of the production machine, the production process implemented by the production machine, and the customer requirements. The following Section 3 will detail these requirements for the execution of engineering model transformations on different platforms.

3 Requirements for Engineering Model Transformations

Reconciliation of engineering models as described in the application scenario in Section 2 can be executed on different model reconciliation platforms depending on the engineering phase in the machine development process (see Figure 12). The term reconciliation is used in this thesis for the requirement to bring engineering models of different engineering disciplines in a consistent state. Model reconciliation can be realized by different technologies with model transformations being one these technologies. This thesis uses model transformations as the technology to achieve model reconciliation.

An Electrical Engineer starts the model reconciliation on his desktop between locally installed engineering applications, if he is working in a small organization. If he goes out for commissioning, reconciliation might be required on his field programming device if he changes the electrical configuration of the machine due to commissioning issues. Finally, a reconfiguration of the machine as part of the operation of the production process can be executed on field programming devices with desktop engineering tools similar to the model reconciliation executed in the commissioning phase.

For larger organizations and more complex systems, engineering models are managed by product lifecycle management (PLM) systems running in a server environment. The PLM server is accessed by many users from multiple engineering disciplines. In contrast to the desktop scenario, a single engineering model like the electrical engineering model is accessed and modified in parallel by multiple users. Generating a consistent machine configuration is part of the PLM workflow within the engineering process of a machine. The model reconciliation required for such a consistent machine is executed on the PLM server. The reconciliation can be triggered on a regularly base (e.g. every night similar to nightly software builds) or based on events (e.g. an electrical engineer commits his working copy of the electrical engineering model).

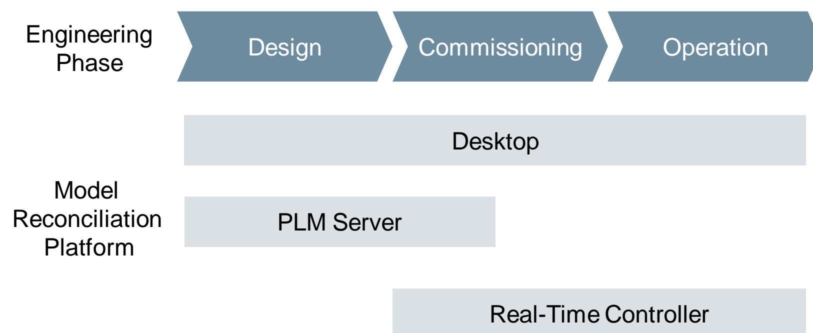

Figure 12: Model Reconciliation of Production Machines

While the previous reconciliation platforms, desktop and server, execute on typical IT system, the third reconciliation platform required for production machine engineering are real-time controllers as used for the operation of production machines. This reconciliation of engineering models is required for on-site reconfigurable machines. The reconciliation of engineering models on real-time controllers can be triggered by machine operators through the human machine interface or by machine operations, e.g. by physically connecting a labeling device to the labeling machine.

3.1 Mission Statement

The reconciliation of models is defined by model transformation specifications within the work of this thesis. "A [model] transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language." [KWB03]

As outlined in the previous section, model transformation rules used by production machines must be executable on different execution platforms. A significant amount of these model transformation rules can be shared by all three model reconciliation platforms. Therefore, a common specification of the model transformation rules is required, which can be executed on different model transformation execution platforms.

For machine vendors, such a platform independent set of model transformation rules ensures independence of the vendors of the machine equipment. The model transformation rules can be executed on automation systems of different vendors or on different system configurations (e.g. on an automation controller or an on a SCADA system alternatively). Therefore, a machine vendor can select the equipment of the machine depending on the customer requirements without the need to rewrite the model transformation rule specification.

Within the industry business, it is also common to deliver the engineering data created by the machine vendor to the customer who operates the machine. In this case, the customer can modify the machine e.g. for maintenance purposes independently from the machine vendor. The model transformation rules required for model reconciliation are part of the engineering data of the machine and must be executable within the customers engineering environment if the machine will be modified. Therefore, the model transformation rules must not only be independent from the machine equipment but also from the engineering environment.

Finally, platform independent model transformation rules allows for the scaling of the engineering environment of machine vendors. Starting from a desktop engineering environment, engineering data including model transformation rules can be transferred to PLM systems if the engineering organization grows.

With these requirements in mind, the mission of this thesis is to define platform independent model transformation rules and the transformation of these platform independent model transformation rules to different model transformation rule execution platforms as shown in Figure 13.

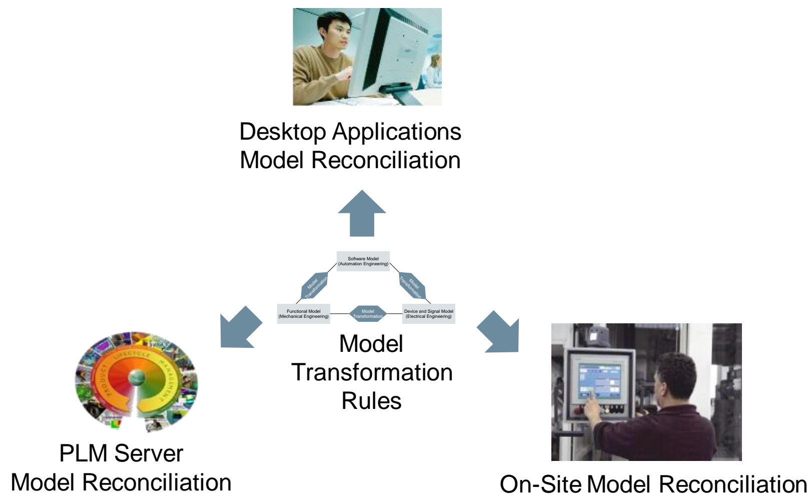

Figure 13: Platform Independent Specification of Model Transformation Rules

3.2 Model Driven Specification of Model Transformations

For the transformation of platform independent model transformation specifications to different execution platforms the model driven architecture (MDA) is used. The model driven architecture as described by the MDA guide [MM03] assumes that a system is built by the iterative application of the MDA. Within a model driven development process, the MDA pattern describes the iterative development of an implementation as the transformation from a platform independent model (PIM) to a more platform specific model (PSM) (see Figure 14). This pattern reflects the development process from a requirement model through concepts and detailed model to the final implementation model. In general, the transformation from a PIM to a PSM requires additional information, such as parameters for the mapping of a PIM to different implementation platforms. This information is shown as additional information in Figure 14 and might involve also entering information by a user before the transformation is started. The new application area of the PIM-PSM pattern presented by this thesis considers the transformations used by the MDA also as models and transforms platform independent models of model transformations (PIM-MT) to platform specific models of model transformations (PSM-MT).

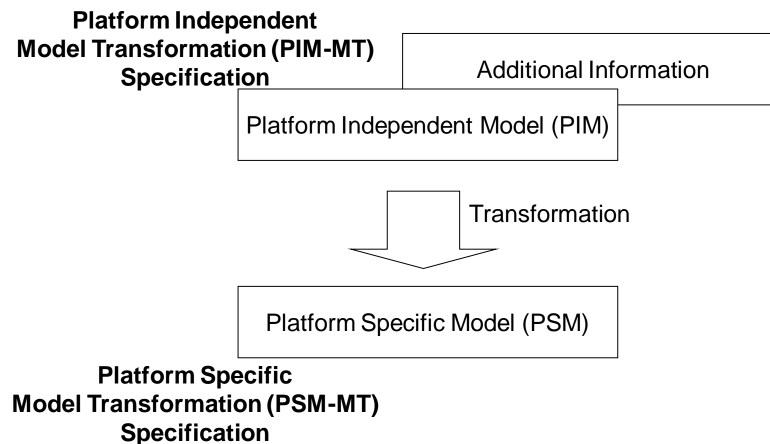

Figure 14: The MDA model transformation pattern [MM03]

The MDA pattern used in this thesis (see Figure 14) involves three different actors as shown in the use case diagram of Figure 15: the automation system provider, the machine builder (OEM), and the machine operator.

The automation system provider specifies the transformation from the platform independent model transformation (PIM-MT) specification to the platform specific model transformation (PSM-MT) specification. The model transformation engines used for this transformation as well as the model transformation engines for the PSM-MT of engineering models are in general standard model transformation engines. Only for real-time model transformations on programmable logic controllers (PLC), a model transformation engine must be developed by the automation system provider, since such a model transformation is not available up to now.

The machine builder (OEM) specifies the platform independent model transformations (PIM-MT) required for the reconciliation between engineering models. He executes the PIM-MT to PSM-MT model transformation provided by the automation system provider to get the desired platform specific model transformation specification (for one of the three platforms shown in Figure 13). The machine builder uses the platform specific model transformation specification (PSM-MT) as part of the machine engineering process to reconcile the engineering models of his machine engineering environment.

Real-time model transformation specifications are not only executed by the machine builder (OEM) within machine test and commissioning, but are also executed by the machine operator as part of the machine reconfiguration for new production orders.

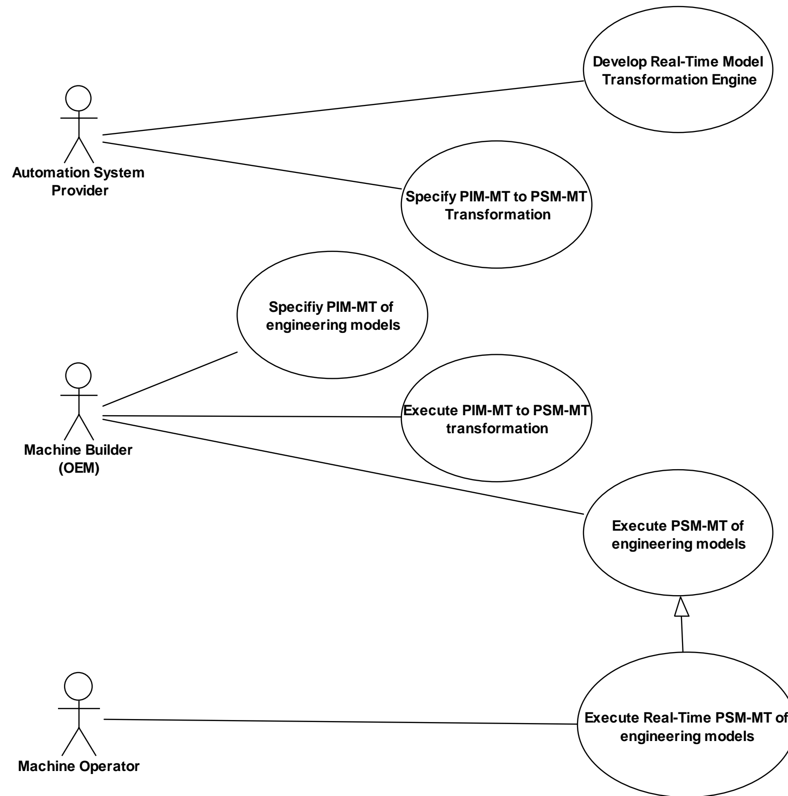

Figure 15: Actors and Use-Cases of Engineering Model Transformations

3.3 Requirements for Engineering Model Transformation Specifications

The key requirement of machine builders for model transformation specifications as presented by this thesis is the platform independence. The same model transformation specification shall be executable on different model transformation platforms depending on customer requests and project needs. The platforms currently relevant for machine builders for the transformation of engineering models are desktop workstations, PLM servers, and real-time automation systems.

Being part of a machine engineering and machine configuration workflow, model transformation must always refine existing models and do not create completely new models. The support of the model refinement is platform specific and might be implemented different on multiple platforms.

The execution of model transformations shall be triggered by user interaction or by automatic processes. User triggers for model transformations are required for machine

engineers in their daily engineering work to update their working model with changes made in engineering models by other users.

Automatic triggers are required for model transformations executed on PLM servers or on real-time automation systems. A PLM server requires the execution of model transformations if a valid configuration of the engineering models is requested, e.g. for machine commissioning or for the generation of the machine documentation. The generation of valid configurations is usually part of the release management within machine engineering. For real-time systems, a valid engineering model configuration is required if the machine is switched from manual mode to automatic mode. Model transformation engines must be compatible to the cyclic execution model of real-time systems. In contrast to the classical desktop usage of model transformation engines, the model transformation is not triggered by discrete events but executed continuously by the real-time system of the automation controller.

Machine engineering is a heavily customized process with respect to the content of engineering models and the data exchange between engineering disciplines. The customization of engineering models is different between companies and evolves over time within a company from machine project to machine project. Together with changes of the engineering process and changes of the engineering models, the model transformation rules must evolve over time. The knowledge about engineering model consistency is part of the engineering knowledge of machine engineers. Therefore, model transformation rules must be easy understandable and modifiable by machine engineers. This requires an easy understandable model transformation specification together with an easy understandable engineering model.

The model transformation rules are part of the engineering data of a machine project. In a managed engineering environment, the model transformation rules must be stored together with the other machine related information on a PLM server or on a version control system. These repositories can easily handle textual information. Therefore, a textual representation of model transformation rules is preferred. A textual representation also eases the exchange of model transformation specifications between different platforms.

In machine engineering models, object oriented models are still very uncommon. For example, real-time automation controllers are configured by key-value lists or electrical engineering models consist of a bill of material together with a cross reference list. Therefore, engineering model transformations must support such weak-typed models within model transformation rules. In a weak-typed model no common classification scheme exists, but classification of elements is implemented by different attributes. Another consequence of weak-typed models is that a strong support for string handling must be provided by the model transformation specification. Beside type classifiers for elements, also type classifiers for attributes are missing. Textual strings are the least common denominator between attribute values of different engineering model elements. Finally, references play a weaker role in machine engineering projects than in general object oriented models. Although references must be valid for a model release (i.e. referencing an object, which exists in the same model as the reference), working

versions of the engineering models can include invalid references due to changes of the machine model (i.e. referencing an object, which does not exists in the same model as the reference). Therefore, the model transformation specification must accept invalid references. References in machine engineering models are for example implemented by common attributes like communication addresses, memory addresses, or reference designators.

The engineering models used on different platforms by model transformations must be extendable to support the adaptation to model transformation engines. For example, meta-information about model elements required by a model transformation engine must be added to the engineering model if the information is not available in the engineering model yet. Such an extension was required for the implementation of model transformations on IEC 61131 programmable logic controllers.

3.4 Structure of Engineering Model Transformation Specifications

The structure of engineering model transformation specifications, as covered by this thesis, is shown in Figure 16. An engineering model transformation specification can be used to run an engineering model reconciliation, e.g. between an electrical engineering model and an automation software model as described in Section 2. In general, multiple engineering model transformation specifications will be used for different reconciliation scenarios. Model reconciliation is triggered by a user working on an engineering model or by the engineering model management environment (e.g. a PLM server) on a regular basis.

An engineering model transformation specification consists of one or more transformation rules. Each transformation rule checks a relationship between some source model elements and target model elements for validity. If the relationship is invalid (usually target elements do not exist or have wrong attributes), the engineering model is modified so that the relationship between the source model elements and the target model elements is valid.

A transformation rule uses model patterns to check the relationship between model elements in a source and a target model. In the application example of the bottle labeling machine used in this thesis (see Section 2.3), a transformation rule defines for example that a label feeder element must be activated in the software model (the target model) if a labeling device drive controller is active in the device and signal model (the source model). The source model pattern selects a set of model elements. In general, the specification of source model elements is not unique with respect to the properties of the source elements. Therefore, a source model pattern selects multiple occurrences in the engineering model. A selection of a source model pattern occurrence is called context for the execution of a model transformation rule. The relationship between source model elements and target model elements is checked based on the current context of the model transformation rule. The target model elements are created and modified based on the information from selected source model elements, for example the name of target model elements can be derived from source model elements or properties of target model elements can be calculated from source model elements.

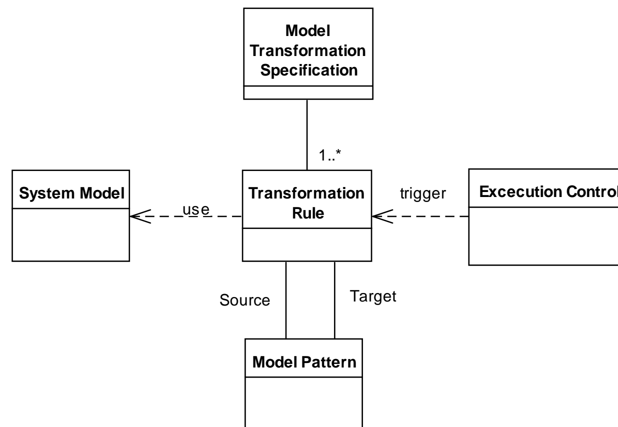

Figure 16: Model Transformation Specification Structure

The access of transformations rules to model elements is specified according to a system model used by the transformation rules. The system model provides the vocabulary, which can be used by the transformation rules to specify model patterns. One part of the vocabulary are classifiers, which can be used to identify model elements (e.g. names of elements type or names of element properties). Another part of the vocabulary describes the model structure: which elements can be referenced from the context of another model element (e.g. which properties belong to a model element and can be read or modified from a reference of this model element).

Finally, the execution control of transformation rules specifies the execution order and execution activation of transformation rules. This part of the model transformation specification is in many cases not explicitly specified but implicitly part of the relationship of model elements, which are created by model transformation rules. An explicit specification of the execution control can be part of a state machine running the model transformation rules or can be specified as attributes of the transformation rules such as an execution priority or rule application conditions.

3.5 Engineering Model Access

The execution of engineering model transformations as described in Section 3.4 requires appropriate interfaces to the engineering models to select and modify model elements. The engineering model transformations use different operations on source model elements and on target model elements: source model elements are only queried but never modified while target model elements are only created or modified but never queried by the target pattern specification. Therefore, source model elements and target model elements have different interfaces as shown in Figure 17. The interface to the source model supports the selection of elements and querying element properties. The target model interface supports the creation of elements and the modification of element properties.

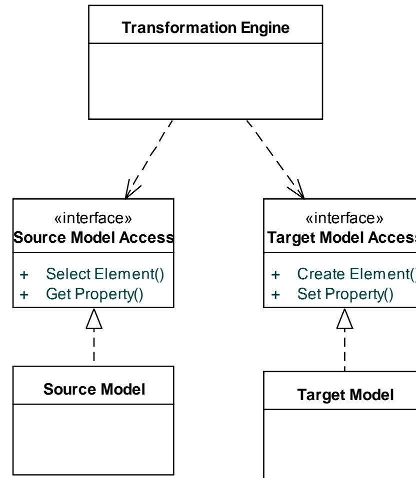

Figure 17: Model Access Operations

Elements, which are accessed by the engineering model operation, are described by the metamodel shown in Figure 18. The metamodel is very small to be adaptable to a large number of engineering models available. The elements of this metamodel are prefixed with EM (for Engineering Model) to distinguish them from elements with similar names in other metamodels.

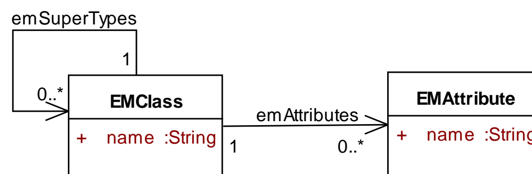

Figure 18: Engineering Metamodel

The model driven approach selected by this thesis requires the classification of elements according to EMClass types (at the left side of Figure 18). Each EMClass type can be identified by a unique name. The engineering information, which can be stored in an instance of an EMClass type, is held by attributes defined by EMAttribute (at the right side of Figure 18)), which are also identified by a unique name. In general, it is difficult to define common data types between different engineering models and between implementations of an engineering model on different platforms. Therefore, the EMAttribute does not include a type specification. The reuse of common model patterns between transformation rules is supported by the emSuperTypes relationship between EMClass types. The emSuperTypes relationship creates two constraints on the EMClass types. The first constraint allows that two EMClass types, which share a common EMClass by an emSuperTypes relationship can be used interchangeable in the model

pattern of a model transformation rule. The second constraint defines that the source EMClass type of an emSuperTypes relationship must hold all EMAttribute data elements of the target EMClass type.

The engineering metamodel used in this thesis does not use an explicit definition of a reference between two EMClass elements. In most metamodels used for model transformations, references build a strong relationship between elements, which may not be violated within a model instance. In engineering models, it is very common that the model is inconsistent due to parallel changes of different people working on the model. Therefore, a weak reference between elements is required for engineering models. Such a weak reference is implemented by attributes which adhere to a naming scheme agreed between the users of different model. Examples for such an agreement are reference designators for industrial equipment [IS09].

Existing engineering models do not provide a common engineering metamodel and a common interface for model operations as presented in this section. Moreover, most engineering models are not designed and prepared to be modified by a model transformation engine. Therefore, an adaptation of the structure of engineering models and the interface to the model transformation are required as shown in Figure 19.

The adaptation of the engineering model structures maps the elements of the engineering model to the metamodel elements shown in Figure 18. First, the elements of the engineering model must be classified in EMClass elements. In general, currently unrelated elements of the engineering model (e.g. drive elements and IO elements for an electrical model) must available as common EMClass types. Then a naming scheme for these classified elements must be defined and unique names must be assigned to each classified engineering model element. These unique names must be automatically be derivable from the engineering model elements, e.g. from properties of the engineering elements or by the addition of additional information to the engineering model.

After the classification of EMClass elements, the available properties of these elements must be defined as EMAttribute elements. The EMAttributes can play different roles in the engineering model. They can be identifiers, which define the position of an EMClass instance in a hierarchical structure within the engineering model, they can be references to another EMClass instance, or they can be ordinary values, which define the parameters of an EMClass instance. The value of an EMAttribute used as an identifier is often derived dynamically from the structure of an engineering model and not stored statically by an EMAttribute. References to other engineering model elements stored in an EMAttribute may be invalid (e.g. if the referenced element is deleted or renamed after the creation of the reference). This implementation of weak references supports engineering workflows which assume that engineering models are only consistent at specific points in the engineering workflow.

The interface adaptation includes the adaptation of source model operations and target model operations. The source model operations "select element" and "get property" must be mapped to common accessible data structures independent of the type of model element. Model transformation engines assume that queries to the source model are

model element independent. Therefore, a common interface must be provided that executes queries either dynamically by dispatching the queries to different engineering model elements or based on a data structure, which stores all information about available model elements.

The creation of elements as required by the target model might be mapped to different operations in the target engineering model. For example, a real-time automation controller does not allow for the creation of elements similar to a new operation in the Java programming language. Instead, all elements that might be available in any automation controller configuration must be available preconfigured on the controller to optimize the memory layout. The creation or deletion of an engineering model element means the activation or deactivation of an already existing model element on this real-time controller. Another implementation of the create operation is required for the refinement of engineering models: after the creation of an element, a merge operation with the already existing engineering model must be executed to avoid duplicate model elements within the reconciled engineering model. This implementation of the create operation is for example required for enterprise model transformation engines, which implement event driven model modifications.

The modification of EMAttribute elements within an engineering model results in different operations similar to the role of EMAttributes within the target model. An EMAttribute, which identifies the position of an engineering model element within the hierarchical structure of target model must be handled together with the create operation to create the model element at the appropriate position of the engineering model. For EMAttributes representing references no special handling is required since invalid references are allowed within the target model by design.

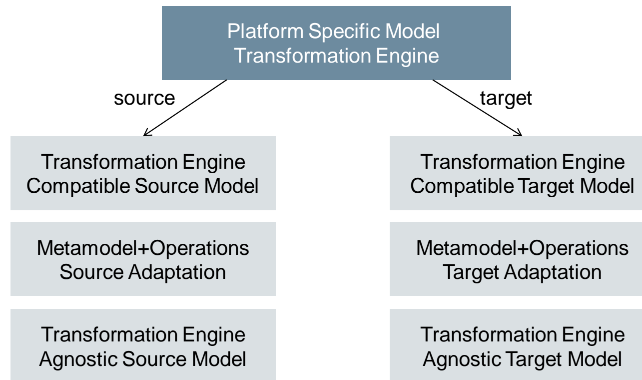

Figure 19: Transformation Engine - Model Adaptation

3.6 Related Work

The MDA manifesto [Bo04], published by IBM Rational Software, promoted the model driven architecture (MDA) as the next level of software engineering building on modeling standards like UML (Unified Modeling Language) [IS12a] and MOF (Meta Object Facility) [Ob11]. The focus of the MDA manifesto was the automated construction of software applications from models based on standard modeling technologies. This approach did not gain much acceptance due to the wide gap between abstract UML models and the complex source code of programming languages like C++. Therefore, the approach presented by this thesis does not focus on code generation for production machine applications but on the reconciliation between engineering models on a similar level of abstraction.

The research roadmap for the model-driven development of complex software systems provided by [FR07] presents three major challenges for the successful application of model driven development: modeling language challenges, separation of concern challenges, and model manipulation and management challenges. These challenges have been tackled by this thesis building on industry technologies. Existing modeling languages were used to realize the concepts developed by this thesis. The separation of concerns is based approved engineering workflows for production machine engineering. The existing engineering environment of production machine builders is used for model manipulation and management.

Instead of the transformation of a platform independent model transformation specification (PIM-MT) to a platform specific model transformation (PSM-MT) as presented by this thesis, a modular model transformation environment could be developed. A modular model transformation environment could be adapted to different execution platforms as required by the production machine engineering process by the replacement of components like the meta-modeling technology or the model transformation language. For example, the Epsilon family of languages and tools [Ko13] allows for the replacement of the modeling technology but is tightly connected to the Java platform for the execution of the model transformation language. A framework, which allows for the execution of a model transformation specification on different platforms, is not available up to now. Therefore, the MDA approach realized by this thesis is required to fulfill the need of production machine builders for the execution of model transformation specifications on different platforms.

4 Platform Specific Model Transformation Languages and Engines

The building blocks of model transformation platforms considered by this thesis are shown in Figure 20. A model transformation engine does not operate directly on a system (e.g. a system like the bottle labeling machine used as an example in this thesis) but on models representing different aspects of a system (e.g. a model of the electrical configuration of a production machine or a model of the software configuration of the production machine). Therefore, the transformed system shown in the lower part of Figure 20 is not part of a model transformation platform. The mechanisms of reflecting changes of the transformed system within the model and vice versa are implementation specific and outside the scope of this thesis. For example, the electrical model of a bottle labeling machine can be updated by communication protocols used for the detection of electrical devices on a field bus.

The upper part of Figure 20 shows the building blocks of model transformation platforms used within this thesis:

- A system model, which is modified by the execution of a model transformation.
- A system metamodel, which defines the structure of valid system models.
- A rule language for the definition of the mapping between models.
- A model transformation language metamodel, which defines the structure of valid model transformation specifications.
- A transformation execution engine, which runs in a specific runtime environment.
- A transformation execution environment which is used to run the transformation execution engine.

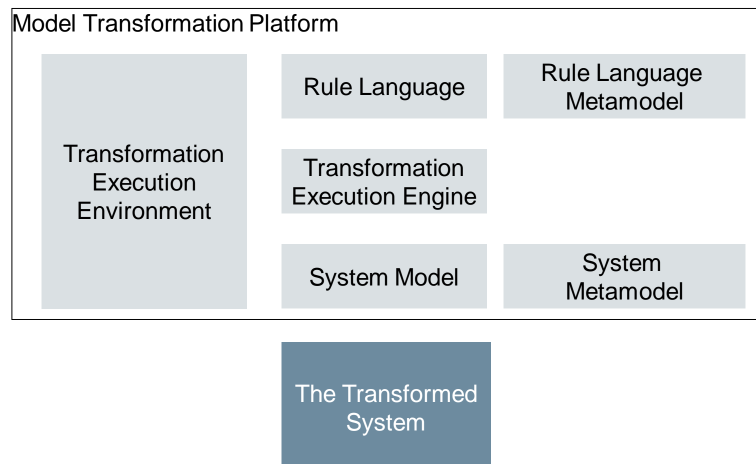

Figure 20: Model Transformation Platform

Each model transformation system is characterized by a specific setup for each of these building blocks. This set of characteristics of a model transformation is immutable for most of the currently available model transformation systems. For example, the model transformation language of an existing model transformation engine can't be modified or substituted. The model transformation systems are designed for a specific transformation execution environment. Therefore, these model transformation systems are considered as platform specific model transformation systems (PSM-MT) within this thesis.

In this section, for each of the three model reconciliation platforms introduced in Section 3, desktop, enterprise PLM server, and real-time controllers, the structure of the model reconciliation platform is analyzed. For platforms, which are only partly or not at all prepared for the execution of model transformations, the required extensions for model reconciliation developed as part of this thesis are presented.

To evaluate a model reconciliation platform, a common evaluation scheme was created as part of this thesis. If a new transformation specification for the transformation from the existing platform independent model transformation specification (PIM-MT) to a new platform specific modeling transformation specification (PSM-MT) platform shall be developed, the following aspects must be considered and mapped:

- rule language
- system model
- pattern language
- inter-rule execution control
- modularization

The rule language defines the transformation from elements of the source engineering model to elements of the target engineering model. Model transformation systems with declarative rule definitions (which express the logic of the transformation without control flow) are easier to handle by PIM-MT to PSM-MT model transformations than imperative rule definitions (which include the control flow), because only the rule logic must be transformed and the control flow can be handled by the target PSM-MT. Therefore, model transformation systems with declarative rule definitions were preferred within this thesis if available for a model transformation platform.

The system model represents the current state of the transformed system. Within the application scenario considered by this thesis, the system model consists of a source model, which reflects the current state of a part of the transformed system (e.g. the device and signal model of the labeling machine), and a target model (e.g. the software model of the labeling machine), which defines a modification of the transformed system required for consistency. The system metamodel describes the elements handled by the rule language within the source and target system models (which are engineering models within the scope of this thesis). The system metamodel describes elements of the application domain, e.g. electrical devices or software function blocks in the application domain of machine engineering covered by this thesis. The representation of the system model as a data structure within the model reconciliation platform must be compatible to the model representation expected by the rule language used for model transformations. If the system model is available in another modeling technology as expected by the

model transformation engine, the system model must be transformed in the representation expected by the model transformation engine. Many model transformation systems use object oriented system model representations, e.g. based on meta-modeling concepts like Ecore [St09]. Therefore, object oriented system models are used within this thesis.

The pattern language is part of the rule language and used to select and modify elements from the system model. For object oriented system models, the pattern language works on typed elements of the system models with a type specific set of attributes. Depending on the design of the system model, the pattern language can either select from a large number of different types with few attributes (called a strong-typed system model) or select from a small number of different types with many attributes with complex values (called a weak-typed system model). For engineering models in machine development, weak typed system models are more common than strong-typed system models. For the usage of weak-typed engineering model as part of a model transformation, the elements of the weak-typed engineering model must be classified according to their attributes and transformed to a strong type system model with a higher number of different types to take advantage of the type support of existing model transformation languages. These types are the primary keys for the selection of elements by the pattern language of a model transformation engine.

The execution of multiple rules within a model transformation definition is coordinated by the inter-rule execution control of a model transformation system. The execution order of model transformation rules can be either implicitly determined by the model transformation engine based on constraints of the system model or explicitly defined by the user.

Finally, the modularization of model transformation definitions eases structuring of the model transformation specification, eases the reuse of definitions of model transformations, and eases the adaptation of model transformations to different engineering systems.

Another scheme for the classification of model transformation approaches was provided by [CH03]. In this classification scheme, the design features of model transformations relevant for the classification of platform specific model transformation specifications (PSM-MT) are transformation rules, rule application strategy, rule scheduling, and rule organization. The transformation rule design feature covers the pattern language considered as one of classification features within this thesis. Inter-rule execution control of this thesis is included in the rule application strategy. The modularization feature is part of the rule organization in [CH03].

The classification of the rule language and the system model as used in this thesis is not part of the feature model used by [CH03]. The taxonomy of model transformation presented by [MV05] uses the term "technological space" for the classification of the system model used by this thesis.

The following subsections analyze the PSM-MT model transformation systems used for desktop, enterprise PLM server, and real-time controllers as part of this thesis. Each

PSM-MT model transformation described here stands as the representative for its kind of model reconciliation platform. The analysis provided by the following subsections can used, if another PSM-MT model transformation system shall be used for a specific setup of a machine engineering environment.

For the real-time PSM-MT as presented in Section 4.3, a new model transformation engine for IEC 61131-3 real-time controllers was developed as part of this thesis. For the other model transformation platforms, desktop and enterprise PLM server, this thesis shows the necessary adaptation of existing modeling transformation platforms for the usage as a PSM-MT for engineering model transformations in Section 4.1 and Section 4.2 respectively.

4.1 Desktop Model Transformation Engine

A desktop model transformation engine is typically executed on the personal computer of an engineer. Before the execution of the model transformation, the engineer must setup the execution environment of the model transformation engine (e.g. by the installation of the transformation engine software package), must transfer the source model from a source engineering application to the local file system of his PC, configure the transformation and run the transformation, and finally transfer the target model from the local file system into the target engineering application (see Figure 21).

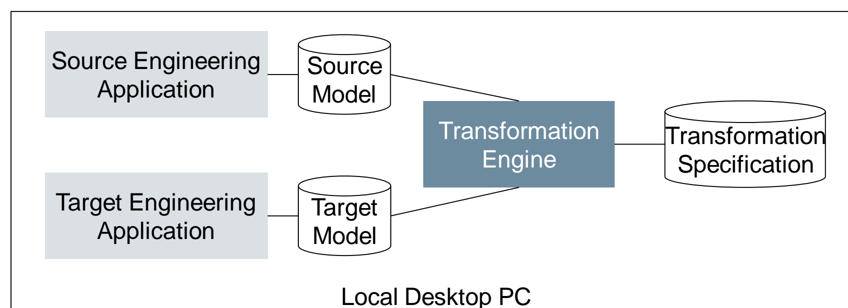

Figure 21: Desktop Model Transformations

The source model and the target model must adhere to a data format which can be used by the transformation engine. Therefore, an adaptation between the engineering application and the model data format must be implemented. In general, such an adaptation is either implemented based on an existing file export from the engineering application (e.g. XML data export) or uses an application programming interface (API) of the engineering tool to generate the data model.

For desktop model transformations, the ATL model transformation language [Jo06] was used in this thesis. ATL was chosen because it provides a mature implementation based on the Eclipse platform[5] with a comprehensive toolset and commercial technical support (e.g. from Obeo[6]). A disadvantage of ATL is that it is available only for the Java programming language and not for the DOTNET environment which is the main programming environment for Windows desktop PC.

[5] http://eclipse.org/atl/
[6] http://www.obeo.fr/

4.1.1 Rule Language

An ATL model transformation is specified by a set of rules, which specify the mapping of source elements to target elements. The ATL rules are aggregated as elements of type ModuleElement in an ATL module, which is the container of all ATL rules belonging to a model transformation specification (see upper part of Figure 22). The declarative ATL rules used as platform independent model transformation rules are a special Rule called MatchedRule (see lower part of Figure 22).

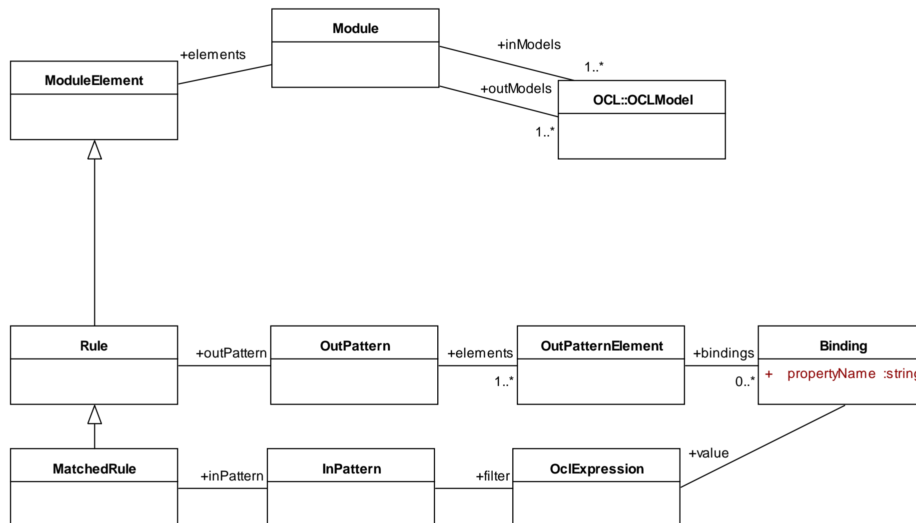

Figure 22: ATL Model Transformation Specification

Each MatchedRule consists of an InPattern and an OutPattern. The InPattern is specified by an OclExpression. The object constraint language (OCL) was originally designed "to describe expressions on UML models" [Ob10]. ATL reuses the OCL type system and OCL declarative expressions as part of pattern definitions. The OutPattern is specified by a set of bindings. Bindings initialize attributes of the created target model elements with the help of OCL expressions.

4.1.2 System Model

The ATL model transformation engine operates on system models based on the Ecore metamodel [St09]. The core elements of an Ecore model are shown in Figure 23: classes, attributes, data types, and references. The engineering metamodel presented in Section 3.5/Figure 18 is easily adaptable to this metamodel, since the engineering metamodel definition used in this thesis is a subset of the elements in the Ecore metamodel. As already mentioned, the engineering metamodel does not use strong-typed attributes and does not use Ecore references to ease the integration with weak-typed models of machine engineering tools. Instead, references are implemented as attributes holding identity values from referenced objects. This approach limits the introspection capabilities of model instances but fits better to the current design scheme of machine engineering models.

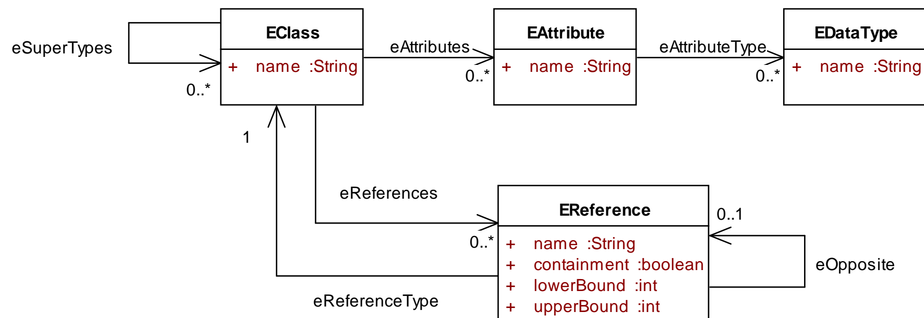

Figure 23: The Ecore kernel [St09]

4.1.3 Pattern Language

The ATL transformation language uses the OCL language [IS12b] for the definition of patterns for querying model elements and modifying model elements. Model queries are only allowed on the source model, while model modifications are only allowed on the target model. Model modifications include the creation of elements and the manipulation of attributes. The deletion of model elements is in general not part of ATL transformations, since the target model is usually generated completely from scratch.

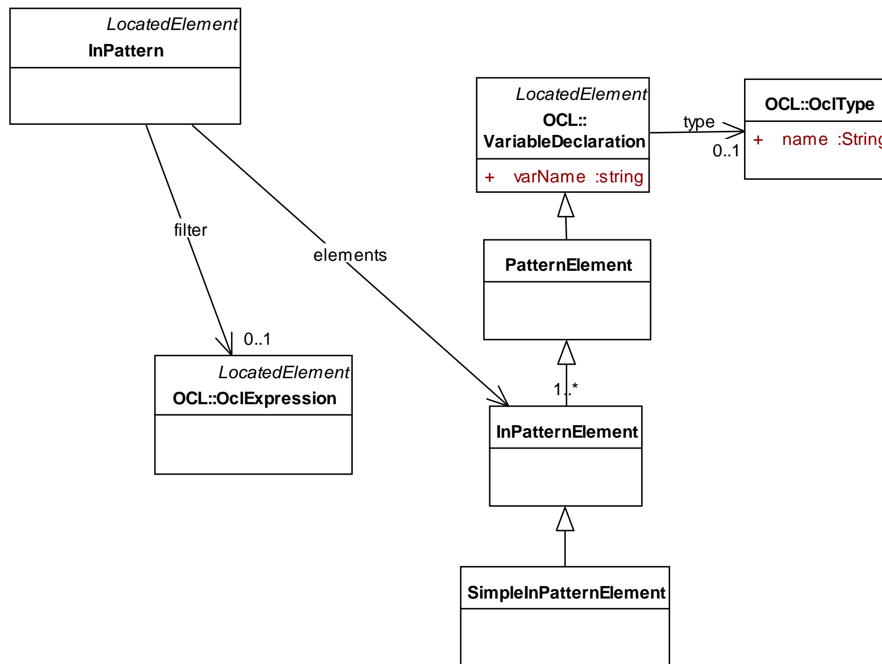

Figure 24: ATL model queries: InPattern

Patterns used for model queries are called source pattern or InPattern in the ATL metamodel (see Figure 24). Target patterns or OutPattern in the ATL metamodel (see Figure 25) are used for model modifications. Both patterns refer to typed elements of the system models.

The InPattern refers to a single typed element of the system model with an assigned variable name. In the example in Figure 26, the type of the element from the system model is "MMSIMOTION!DriveObject" and the assigned variable name is "s". The second part of an InPattern is a filter condition, which is formulated as an OCL expression. Commonly used filter conditions are filters for the instance name as shown in Figure 26, filters for the position within the hierarchical structure of an engineering model (e.g. a path to the instance), or specific attribute values (e.g. the logical address of an automation device).

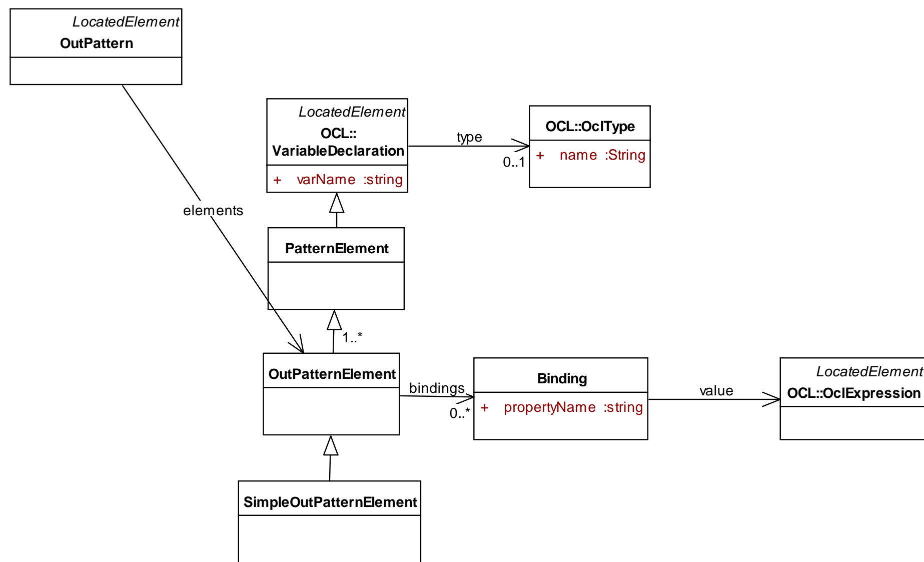

Figure 25: ATL model modifications: OutPattern

The OutPattern of an ATL rule creates one or more instances of system model elements within the target model. Each instance has an assigned type from the system model and is identified within the rule by a variable name. This variable can be used by subsequent OutPatternElements to refer to attributes. The OutPatternElements are created within the order of their definition within the ATL rule definition. Therefore, it is not possible to reference from an OutPatternElement to attributes from a subsequent OutPattern element (e.g. it is not possible to reference attributes of t from u in Figure 26) but attributes can only be referenced from preceding elements. The definition of attributes and references as part of the creation of elements is called binding in the ATL transformation language (see Figure 25). Similar to the definition of filters in ATL InPatterns, bindings are defined by OCL expressions.

```
-- @path MMELECTRICAL=/pim_mt/model/electrical.ecore
-- @path MMAUTOMATION=/pim_mt/model/automation.ecore

module electrical2automation;
create OUTAUTOMATION : MMAUTOMATION
	from INELECTRICAL : MMELECTRICAL;

helper def :
 RDLabelDeviceControl(driveObject : MMELECTRICAL!DriveObject)
 : String = 'fbrd_'+driveObject.name;

helper def :
 RDTechnologyObject(driveObject : MMELECTRICAL!DriveObject)
 : String = 'tord_'+driveObject.name;

rule do2to
{
	from
		s: MMELECTRICAL!DriveObject
	to
		u: MMAUTOMATION!LabelDeviceControl
		(
			name <- thisModule.RDLabelDeviceControl(s)
		)
		, t: MMAUTOMATION!TechnologyObject
		(
			name <- thisModule.RDTechnologyObject(s)
		)
}
```

Figure 26: ATL rule example with InPattern and OutPattern

4.1.4 Inter-Rule Execution Control

ATL does not support explicit control of the order of rule execution for matched rules. ATL matched rules are executed in the order of their definition in the ATL module. In the example in Figure 27, the rule do2to is executed before the rule cu2control. For the ATL model transformation, the rule order is less important, because the execution of an ATL model transformation consists of two phases. In the first phase, all rules are executed and traceability links are created for information that must be exchanged across rules (e.g. cross rule attribute values or cross rule references)[YW09]. In the second phase, the missing information of all generated model elements (e.g. unresolved references to created target model elements) is added with the help of the traceability links created in the first phase of the ATL model transformation execution.

```
module electrical2automation;
create OUTAUTOMATION : MMAUTOMATION
       from INELECTRICAL : MMELECTRICAL;

rule do2to
{
       from
              s: MMELECTRICAL!DriveObject
       to
              u: MMAUTOMATION!LabelDeviceControl
}

rule cu2control
{
       from
              s: MMELECTRICAL!DriveControlUnit
       to
              t: MMAUTOMATION!LabelDeviceCuControl
}
```

Figure 27: ATL rule execution order

Beside matched rules, which are used in the application scenario of this thesis, ATL also knows lazy rules and called rules. These two additional ATL rule types can be called from matched rules. This allows for a limited inter-rule execution control, since the lazy rules and the called rules are executed together with the calling matched rule. ATL lazy rules and called rules are not considered as a cross-platform concept and are, therefore, not used for platform specific desktop model transformation specifications within this thesis.

4.1.5 Modularization

ATL supports two modularization concepts: separation of the system model from the model transformation modules and grouping ATL model transformation rules in different modules.

The first modularization option, separation of model transformations and associated models, is implemented by ATL launch configurations (see Figure 28). A launch configuration allows for the execution of an ATL model transformation (an ATL module) for different source and target models. Even the metamodel of the source- and the target model might be different for multiple launch configurations, if the model elements referenced by the ATL model transformation are still available.

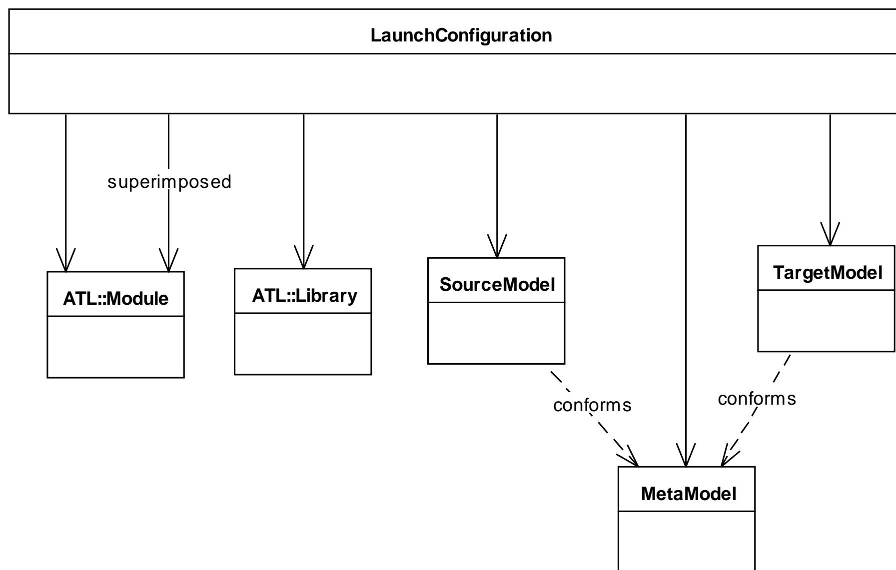

Figure 28: ATL launch configuration

The second modularization option, grouping ATL rules in multiple files, is visible in an ATL launch configuration by superimposed ATL modules and by ATL libraries. ATL libraries allow the reuse of ATL helper methods for multiple ATL transformation definitions. An example of superimposed ATL modules is shown in Figure 29. The first module, "electrical2automation", defines rules specific for the transformation of an electrical model to an automation model. It uses the superimposed ATL module "model2model", which includes rules required for the transformation of common model elements like the root element of an engineering model.

```
module electrical2automation;
create OUTAUTOMATION : MMAUTOMATION
from INELECTRICAL : MMELECTRICAL;
uses model2model;

rule do2to
{
	from
		s: MMELECTRICAL!DriveObject
	to
	u: MMAUTOMATION!LabelDeviceControl
	, technologyObject: MMAUTOMATION!TechnologyObject
}
```

```
module model2model;
create OUTAUTOMATION : MMAUTOMATION
from INELECTRICAL : MMELECTRICAL;

rule model2model
{
	from
		s: MMELECTRICAL!Model
	to
		t: MMAUTOMATION!Model
}
```

Figure 29: ATL modularization by superimposed modules

Both ATL modularization concepts are useful if a platform specific model transformation shall be adapted to different engineering models. The launch configuration can be used to address different models and metamodels, which shall be used for a transformation. The superimposed models allows for the adaptation of ATL transformations to different model content. For example, if specific model elements should not be considered for the execution of a model transformation, the related superimposed ATL modules can be omitted for that model transformation execution.

4.1.6 Implementation Alternatives

For desktop model transformations, an existing model transformation engine was selected according to the requirements presented in Section 3 and adapted for the usage as a PSM-MT engine. Beside the selected ATL model transformation engine, many other desktop transformation engines were developed in academia. Therefore, the open source

platform Eclipse[7], programmed in Java, with its existing Eclipse Modeling Framework (EMF) is the foundation of all relevant desktop model transformation engines. The other major desktop development technology beside Java, the .NET framework[8] on Microsoft Windows, is not well accepted in academia due to its closed source development model and its license cost. Another drawback is the lack of a modeling framework foundation like EMF for Eclipse, which could be used as a prerequisite for a model transformation engine. Therefore, no relevant model transformation engine exists for the .NET platform until now.

The QVT standard [Ob11] was developed by the OMG to unify the model transformation systems similar to the success of the unified model transformation language UML. Beside some announcements of commercial implementations and some academic research activities, the QVT standard didn't gain much acceptance and seems to be abandoned. An implementation of the operational part of QVT was initially provided by Borland[9] and donated to the Eclipse modeling project[10]. QVT Operational is an imperative language for the mapping of between source and target models. MediniQVT[11] provides an implementation of QVT relational, the declarative language defined in the QVT standard.

The Epsilon project[12] provides a framework for model management, including languages and tools for model transformations. This framework is used as a base for research and implementation of new approaches in model transformation. The epsilon model transformation language (ETL) together with the epsilon object language (EOL) [KPP06] are inspired by concepts already introduced by ATL and OCL but try to overcome known limitations of this environment for model management purposes (e.g. OCL limitations when used as a general purpose programming language [KPP06]). For its dedicated purpose, model to model transformations, ATL currently is more focused and mature than Epsilon.

The Tefkat[13] languages is another Eclipse EMF based model transformation environment with declarative rule definitions and an SQL inspired syntax [LS06]. Tefkat was developed in the context of the QVT request of OMG but is not actively developed for several years.

Another approach used by desktop model transformations are graph transformations engines, e.g. based on triple graph grammars (TGG) [Sc95]. The eMoflon[14] implementation of TGG is available as an Eclipse plugin. A distinctive feature of TGG to other desktop model transformation engines is the support of bidirectional transformations. The same model transformation specification can be used to transform a

---

[7] http://eclipse.org/
[8] https://www.microsoft.com/net
[9] http://www.borland.com
[10] http://www.eclipse.org/projects/project.php?id=modeling.mmt.qvt-oml
[11] http://projects.ikv.de/qvt/wiki
[12] http://www.eclipse.org/epsilon/
[13] http://tefkat.sourceforge.net/
[14] http://moflon.org/

source model to a target model as well as to transform the target model back to the source model. For that purpose, different TGG are translated to multiple model transformations defined by story diagrams [ZSW99] for each transformation direction. The eMoflon implementation is still in an initial state and was, therefore, not used by this thesis.

4.1.7 Summary

Desktop model transformation engines provided the first implementations of the concepts outlined by the model driven architecture (MDA) [MM03]. Although, being the first implementations of model transformations, no accepted standard with mature implementations of desktop transformation systems exists until now. Therefore, the ATL transformation language has been chosen as one of the desktop transformation languages with enhanced tool support and many application examples.

Desktop transformation languages rely on strong-typed engineering models and strong references between engineering model elements. This section defined the subset of model transformation specification features required by engineering model transformations with weak relationships between model elements based on reference designators. The system metamodel of engineering model transformations replaces the Ecore specific definition of references by weak references defined by string attributes holding reference designators specific to the transformed system.

Inter-rule execution control is only influenced by the structure of the system model and not by the transformed system, which is modified by transformations of the system model. For example, constraints on the rule execution order given by the current state of a production machine (e.g. locking of parts of the system) cannot be handled by the ATL model transformation specification.

PSM-MT feature	**ATL**
Rule Language	ATL matched rules
System Model	Ecore metamodels
Pattern Language	OCL
Inter-Rule execution control	ATL traceability
Modularization	Rule superimposition and ATL libraries

Figure 30: ATL PSM-MT features

4.2 Enterprise Model Transformation Engine

In an enterprise environment, discipline specific models are not stored in a desktop environment, but as part of a PLM (product lifecycle management) backbone as shown in Figure 31. Usually a desktop system has only a single authoring system installed according to the discipline specific tasks of the user of the desktop system. For example, an engineer using an electrical CAD (computer aided design) application for electrical wiring and electrical device installation usually doesn't have the software engineering environment for automation controllers installed. Nevertheless, the electrical engineer expects that changes of his colleague from the software development department are reconciled with his electrical CAD model to reflect changes in the electrical device configuration. Therefore, not only the discipline specific models are stored as part of the PLM backbone but also the model transformations required to reconcile the discipline specific models must be executed as part of the PLM backbone as shown in Figure 32.

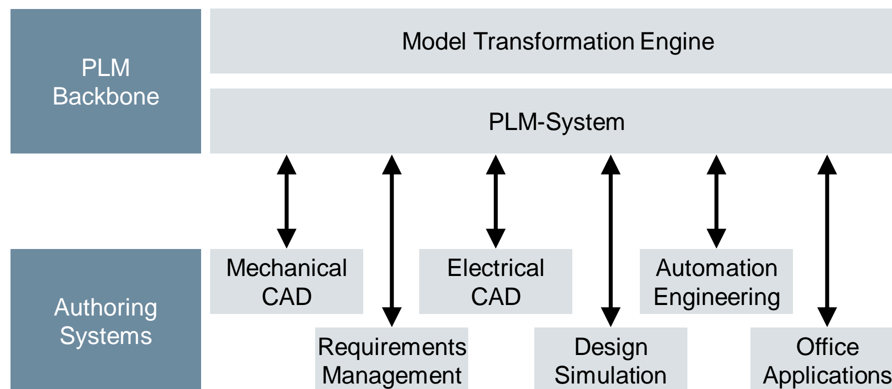

Figure 31: PLM Environment of a Machine Builder [ES09]

The data format used for the integration between discipline specific models is usually XML. Most rule engines for data integration use an architecture based on data connectors to the authoring applications. These connectors get the source model as an XML data representation, apply the defined transformation rules and create an XML data representation of the target model, which is transferred to the authoring tool.

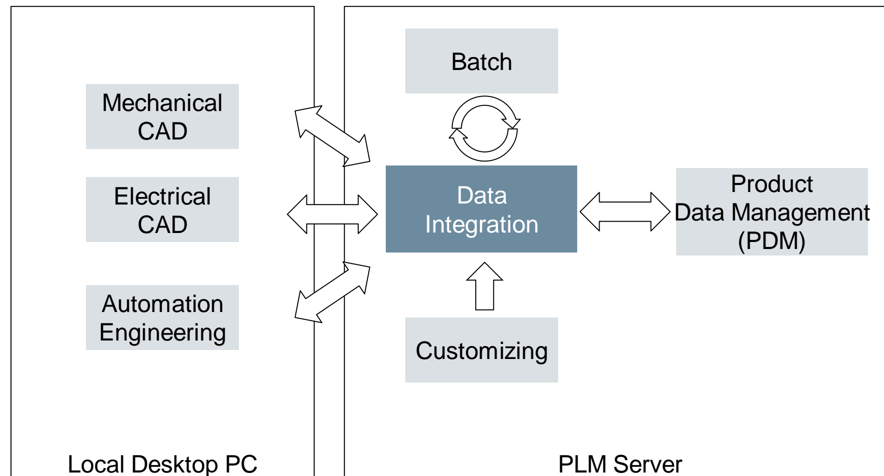

Figure 32: Model Transformations between Authoring Systems [ES09]

For enterprise engineering model transformations, the model transformation engine is part of a PLM server (see Figure 33) and does not run on the local desktop PC together with an engineering application as for desktop model transformations (see Figure 21). The engineering applications used to author the source and target engineering model are still executed on the local desktop PC of engineers. In contrast to desktop engineering model transformations, these engineering applications are executed on different PCs, because they are used by different engineers as part of their discipline specific engineering task.

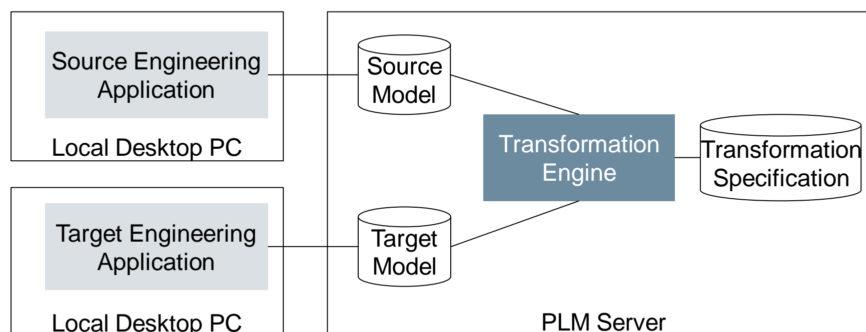

Figure 33: Enterprise Model Transformations

The main technology currently used on enterprise PLM servers for the exchange of engineering model data is the XML language [Wo08] with its related specifications (see Figure 34). The XML language together with the XML schema [Wo12] definition is used for the definition and for the exchange of engineering models.

The transformation of engineering models is defined by XSL transformations (XSLT) [Wo07] and executed by XSLT processors. XSLT is part of the extensible stylesheet language (XSL) family[15]. It is itself an XML language but uses the textual language XPATH [Wo10] for pattern matching.

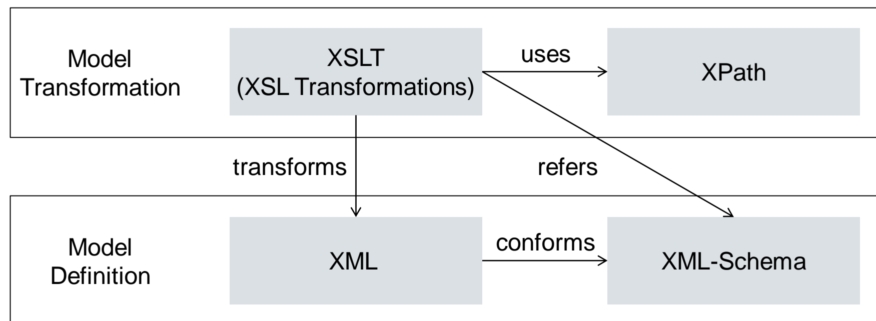

Figure 34: Enterprise Data Transformation Technologies

4.2.1 Rule Language

An engineering model transformation as specified by the XSL transformation language 2.0 [Wo07] consists of a set of template rules included in an XSLT stylesheet (see Figure 35).

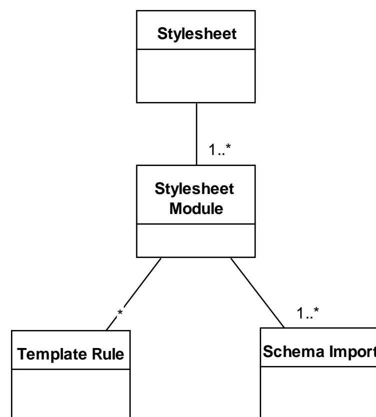

Figure 35: XSL Transformations (XSLT)

[15] http://www.w3.org/Style/XSL/

An XSLT stylesheet is a well-formed XML 1.0 document [Wo08]. An important role of XSLT is the application of styling information to XML source documents for the transformation into presentation formats like HTML or SVG. For that purpose, XSLT is well supported on enterprise servers and webservers for the separation of data and representation. The transformation of XML source documents is not limited to presentation formats but can also be used to transform an XML document representing a source engineering model to another XML document representing a target engineering model. Because of these two features, enterprise server support and engineering model transformations, XSL transformations were used as a platform specific model transformation language within this thesis.

The rule language for engineering model transformations provided by XSL transformations consists of template rules (see Figure 36). An XSLT template rule selects a node in the source engineering model according to a match pattern. For each application of a template rule, a tree for the target engineering model is constructed according to a sequence constructor. The construction of the tree in the target model can use information from the source engineering model by navigating from the selected source node to other source model elements.

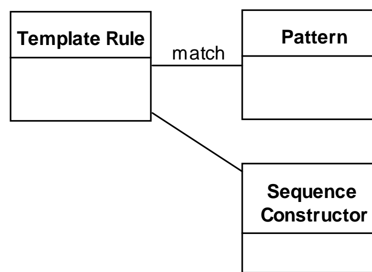

Figure 36: Template Rule

4.2.2 System Model

The system model used by platform specific enterprise model transformations is based on XML files representing engineering models. An XML file consists of elements, attributes, and data. The structure of valid XML documents consisting of these three items is defined by the XML Schema Definition Language (XSD) [Wo12]. An XSD definition itself is an XML document with a schema element as XML document root (see upper part of Figure 37). The XML schema includes two different kinds of declarations: the declaration of elements and the definition of types. An element declaration defines valid occurrences of XML elements and XML attributes with an XML document, which conforms to an XML schema definition. Complex type definitions allow the reuse of the structure of XML elements within a schema definition.

For engineering models used in product lifecycle management (PLM) systems in an enterprise environment, the structure of XML documents representing engineering models must adhere to a schema definition, which can be handled by the PLM system. Therefore, the system model used for engineering model transformations is restricted by a schema definition of the PLM system used for engineering model management (see the middle part of Figure 37). Within this thesis, the SIEMENS Teamcenter PLMXML [Si11a] schema definition was used for the platform specific representation of engineering models within enterprise PLM systems. The PLMXML schema definition is publicly available and is used for the data exchange between the Teamcenter PLM system and other PLM systems or authoring applications.

A common element of models used by product lifecycle management systems is an item:

> "The development of product lifecycle management and the use of different product lifecycle management systems are very largely based on the use of items. An item is a systematic and standard way to identify, encode and name a product, a product element or module, a component, a material or a service. Items are also used to identify documents. What an item means depends upon the specific needs and products of each company." [SI08b]

Within the PLMXML schema definition, an item type can be defined as an extension of the StructureBase complex type defined by the PLMXML schema. The lower part of Figure 37 shows as an example of the definition of a DriveControlUnit and DriveObject as part of the device and signal model (a domain PLMXML specific schema) of the bottle labeling machine application used as an example in this thesis (see Section 2). DriveControlUnit and DriveObject are new item types which are part of the system model of the bottle labeling machine.

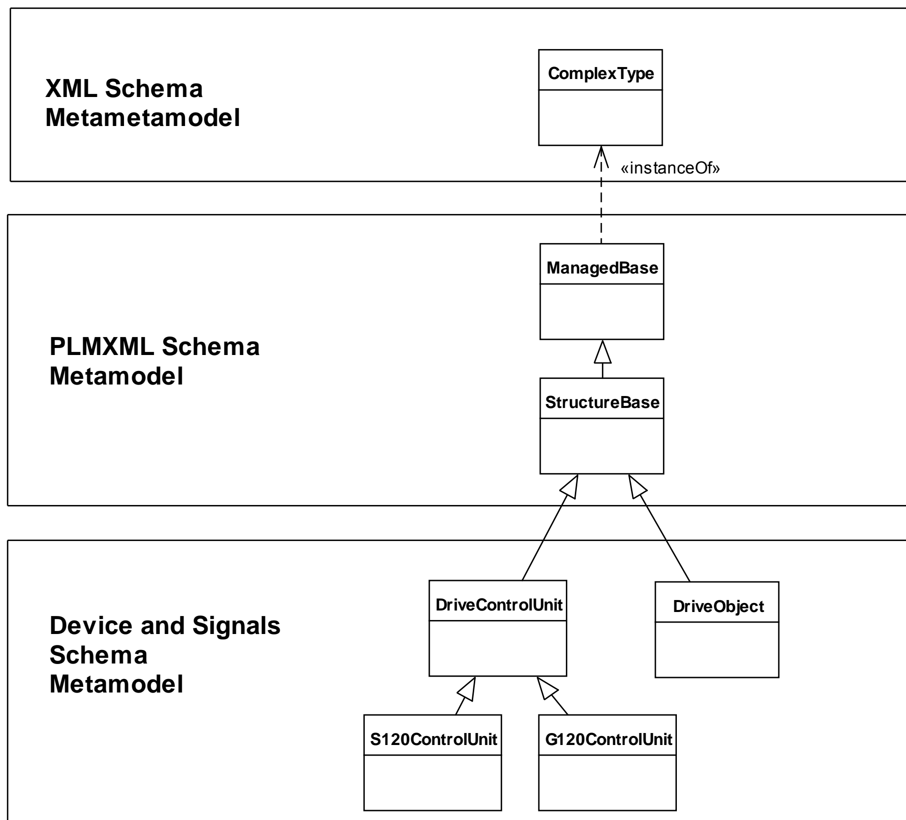

Figure 37: PLMXML schema extension

4.2.3 Pattern Language

The XSLT template rules use different pattern languages for patterns selecting elements from the source elements (the match pattern) and for patterns creating target elements (the sequence constructor, see Figure 38 for both pattern languages).

XSLT template rules use the XPATH 2.0 [Wo10] expression language for the definition of match patterns.

> "Definition: A pattern specifies a set of conditions on a node. A node that satisfies the conditions matches the pattern; a node that does not satisfy the conditions does not match the pattern. The syntax for patterns is a subset of the syntax for expressions." [Wo07]

For usage within engineering model transformations, the match patterns must provide expressions to select element nodes according to their schema type or schema super type. This is called "SequenceType Matching" in XPATH and executed by the element test sequence type matching expression:

> "element(ElementName, TypeName ?) matches an element node whose name is ElementName if derives-from(AT, TypeName) is true, where AT is the type annotation of the element node." [Wo10]
>
> "The definition of SequenceType matching relies on a pseudo-function named derives-from(AT, ET), which takes an actual simple or complex schema type AT and an expected simple or complex schema type ET, and either returns a boolean value or raises a type error [err:XPTY0004]. The pseudo-function derives-from is defined below and is defined formally in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)]:
>
> – derives-from(AT, ET) returns true if ET is a known type and any of the following three conditions is true:
>   1. AT is a schema type found in the in-scope schema definitions, and is the same as ET or is derived by restriction or extension from ET
>   2. AT is a schema type not found in the in-scope schema definitions, and an implementation-dependent mechanism is able to determine that AT is derived by restriction from ET
>   3. There exists some schema type IT such that derives-from(IT, ET) and derives-from(AT, IT) are true." [Wo10]

An example of a match pattern is shown in Figure 39. It selects all nodes of an engineering model with an element type of "DriveControlUnit" or any derived type from "DriveControlUnit" (e.g. S120ControlUnit or G120ControlUnit according to Figure 37). To evaluate type specific match expressions, the schema of the source engineering model must be introduced to the XSLT processor by a schema import instruction (see "xsl:import-schema" at the beginning of Figure 39). The type checking features of XSLT are not available with a basic XSLT processor. They require a schema-aware XSLT processor. Until now, implementations of schema-aware XSLT processors are hardly available.

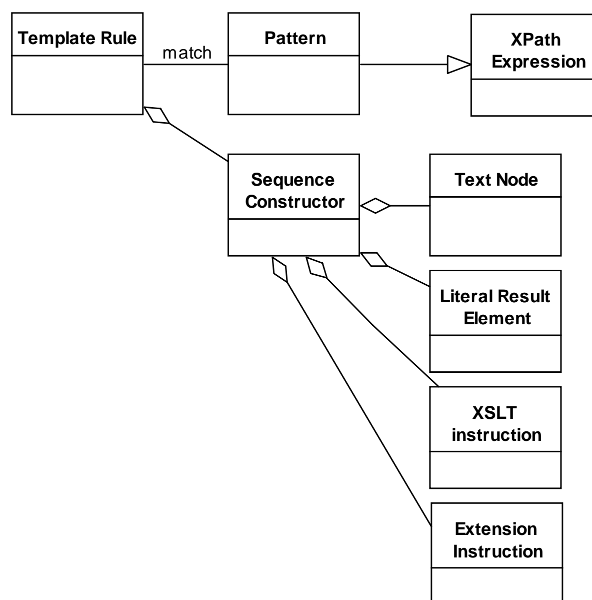

Figure 38: Template Rule Patterns

The sequence constructor of an XSLT template rule is used within platform specific model transformations to create an XML subtree within the target engineering model (the term "sequence constructor" in XSLT 2.0 replaced the term "template" as used in XSLT 1.0). The subtree can be either constructed from fixed template content (by Text Nodes or Literal Result Elements) or can be based on dynamic content (by XSLT instructions or Extension Instructions). XLST instructions are for example used to calculate the value of an attribute in the target engineering model from values of the source engineering model. Extension instructions are user defined functions that can be used for example to provide platform specific implementations of required calculations. The sequence constructor used as an out pattern in Figure 39 creates a "LabelDeviceCuControl" element in the software model of the labeling machine. Its name attribute is set by a user defined function "RDLabelDeviceCuControl" (RD stands for reference designator), which creates valid names of "LabelDeviceCuControl"

elements according to the given name of the "LabelDeviceCuControl" from the devices and signals model. Its "labelDevice" attribute references the name of a "LabelDeviceControl" object within the software model. This name is created by the "RDLabelDeviceControl" function from a reference to a "DriveObject" in the electrical and signal model.

```
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      >

<xsl:import-schema
  namespace="http://www.plmxml.org/Schemas/PLMXMLSchema"
  schema-location="../model/PLMXMLElectricalSchema.xsd " />

  <!-- rule cu2control-->
  <xsl:template match="element(*, plmxml:DriveControlUnit)">

    <!-- OutPattern -->
    <xsl:element name="plmxml:LabelDeviceCuControl"
      type="plmxml:LabelDeviceCuControl">
        <xsl:attribute name="name">
          <xsl:value-of
            select="plmxml:RDLabelDeviceCuControl(@name)"/>
        </xsl:attribute>
        <xsl:attribute name="labelDevice">
          <xsl:value-of
           select="plmxml:RDLabelDeviceControl(@driveObject)"
           />
         </xsl:attribute>
    </xsl:element>

</xsl:stylesheet>
```

Figure 39: XSLT template rule example with match pattern and sequence constructor

4.2.4 Inter-Rule Execution Control

XSLT template rules are executed as part of an XSL transformation starting from an initial template:

> "The transformation is performed by evaluating an initial template. If a named template is supplied when the transformation is initiated, then this is the initial template; otherwise, the initial template is the template rule selected according to the rules of the xsl:apply-templates instruction for processing the initial context node in the initial mode." [Wo07]

The further application order of XSLT template rules after the initial template rule is controlled by "apply-templates" instructions. Used within a template rule, the "apply-templates" instruction executes all XSLT template rules, which match the sequence of nodes given as a parameter to the "apply-template" instruction. The default sequence of nodes of the "apply-templates" instruction causes all the children of context node to be processed.

4.2.5 Modularization

The XSLT language supports the modularization of an XSLT stylesheet in multiple files, which are included in a principal stylesheet module:

> "A stylesheet may consist of several stylesheet modules, contained in different XML documents. For a given transformation, one of these functions as the principal stylesheet module. The complete stylesheet is assembled by finding the stylesheet modules referenced directly or indirectly from the principal stylesheet module using xsl:include and xsl:import elements" [Wo07]

The "include" and "import" XSLT instructions allow for the definition of the precedence of the imported template rules. For imported modules, the template rules of the importing module take precedence over template rules of the imported module. For included modules, the template rules of the included module take precedence over template rules of the including module.

Splitting up an XSLT stylesheet in multiple modules is similar to the modularization concept of the desktop transformation engine ATL with different ATL modules described in Section 4.1.5.

The separation of the system model from the model transformation modules is different for XSLT and for ATL. For XSLT the definition of the system models used for the transformation is part of the XSLT language (the "import-schema" XSLT instruction) while ATL uses an external configuration for the definition of the system models used for the transformation. The modularization provided by the ATL launch configuration is not part of the XSLT standard but might be provided by the implementation of an XSLT processor on an enterprise server. For example, the URI used by the XSLT "include" instruction can mapped from a virtual location to a real module by the XSLT processor. This provides the same benefit as the ATL launch configuration.

4.2.6 Implementation Alternatives

For enterprise model transformations, an existing model transformation engine was selected according to the requirements presented in Section 3 and adapted for the usage as a PSM-MT engine. Alternatively to model transformations by the selected XSLT technology, the terms "enterprise application integration (EAI)" [Li00] or "data integration" [DHI12] are more common on an enterprise level than "model transformations". Within an enterprise application integration solution, the transformation of data as executed by model transformations is only a small part of the infrastructure required for application integration. Workflow management, event handling, or web services are additional components of an enterprise application integration system. Therefore, a model transformation engine is only a part of an enterprise application integration environment.

Microsoft BizTalk Server[16] is an integration solution for business process automation within companies.

> "At its most basic, BizTalk is designed to receive inbound messages, pass them through some form of logical processing, and then deliver the result of that processing to an outbound location or subsystem." [DM07]

The logical processing stage of a BizTalk solution includes the transformation from a source message, which includes a source model instance for engineering model transformations, to a target message, which includes a target model instance for engineering model transformations. Transformations are designed within BizTalk using BizTalk maps [DW09]. BizTalk maps provide a graphical data flow editor, which maps data from a source XML schema to a target XML schema. The graphical mapping representation is compiled to an XSLT representation, which is used by the BizTalk environment for message transformation.

Similar to the BizTalk maps, Altova MapForce[17] is a commercial tool, which defines data transformations by a graphical data flow language from a source to a target schema. MapForce only provides the transformation engine, but not a complete enterprise application environment such as BizTalk. Like BizTalk maps, MapForce generates XSLT from the graphical representation to be used by XSL transformation engines. The transformations created by MapForce can be executed in an arbitrary server environment. For example, the Siemens Teamcenter PLM server can use MapForce transformations to map a source engineering model to a target engineering model.

Both enterprise transformation engines, BizTalk and MapForce, do not provide an external representation of their transformation language, which could be used in MDA workflows for the transformation of a platform independent model (PIM) of a transformation specification to a platform specific model (PSM) of a BizTalk or MapForce transformation as proposed by this thesis.

---

[16] http://www.microsoft.com/biztalk

[17] http://www.altova.com/mapforce.html

The Drools Business Logic integration Platform[18] includes the Drools Expert rule engine, which is based on the Rete algorithm [Fo82]. The Drools platform handles models consisting of Java objects. It can be integrated into enterprise application integration frameworks like Spring [Lu11] or Apache Camel[19]. The models handled by Drools Expert rules are represented by Java beans representing business objects. Therefore, engineering models as considered by this thesis, would need a bidirectional adapter to Java to be usable with the Drools expert engine.

4.2.7 Summary

XSLT as a member of the extensible stylesheet language (XSL) family (beside XSL-FO formatting objects and XPATH [Te05]) was originally not designed as a model transformation language but for the transformation of XML documents to documents with another representation of the original document. With representation of models as XML documents, it is also very common to use XSLT as model transformation language, which transforms an XML document representing a source model in another XML document, which represents a target model. For the transformation of engineering models represented by XML documents, this section showed how XSLT can be used as an enterprise model transformation engine with a similar rule structure as for desktop model transformation engines presented in Section 4.1 if a restricted subset of the XSLT language is used.

In comparison to the ATL desktop model transformation engine introduced in the previous section, XSLT provides more advanced features for pattern handling. For example, XPATH queries allow for the selection of node sets according to declarative queries, while OCL used as an ATL pattern language only allows the navigation relative from a context node. Therefore, user defined functions are used instead of XPATH queries and templates to reduce complexity for the higher order transformation (HOT) of platform independent model transformation specifications (PIM-MT) to platform specific model transformation specifications (PSM-MT) as described later in this thesis.

Handling of element types and extension relationships as defined by XML schema (see Section 4.2.2) is not very common for XSLT transformation engines until now and currently only supported by a few schema aware XSL transformation engines. Schema aware transformations are only available for source patterns. The target patterns are constructed without schema checking by the XSLT transformation engine.

Desktop model transformation engines as described in Section 4.1 usually apply the complete set of transformation rules to a complete engineering model definition. If the same execution strategy is applied to enterprise model transformation engines like XSLT in server systems with middleware like message systems and multi-user handling, the complete set of transformation rules is executed on a subset of the engineering model definition. Therefore, platform specific model transformation rules must be executable on source and target model definitions with invalid references or with invalid attribute

[18] https://www.jboss.org/drools/

[19] https://camel.apache.org/

values. The engineering model specification and transformation rule definition presented in this section takes this account with the special implementation of references by reference designators, which allows for the independent execution of each transformation rule without the need of fixing invalid references in a second transformation phase similar to ATL.

PSM-MT feature	**XSLT**
Rule Language	XSLT template rules
System Model	XML schema definition
Pattern Language	XPATH 2.0 and XSLT
Inter-Rule execution control	"apply-templates" instructions
Modularization	XSL stylesheet modules

Figure 40: XSLT PSM-MT features

4.3 Real-Time Model Transformation Engine

Real-Time engineering model transformations, as considered by this thesis, are executed on programmable logic controllers (PLC) on production machines (see Figure 41). Programmable logic controllers as defined by IEC 61131-1 [In03a] are used in industrial environments to control production devices such as machines and plants.

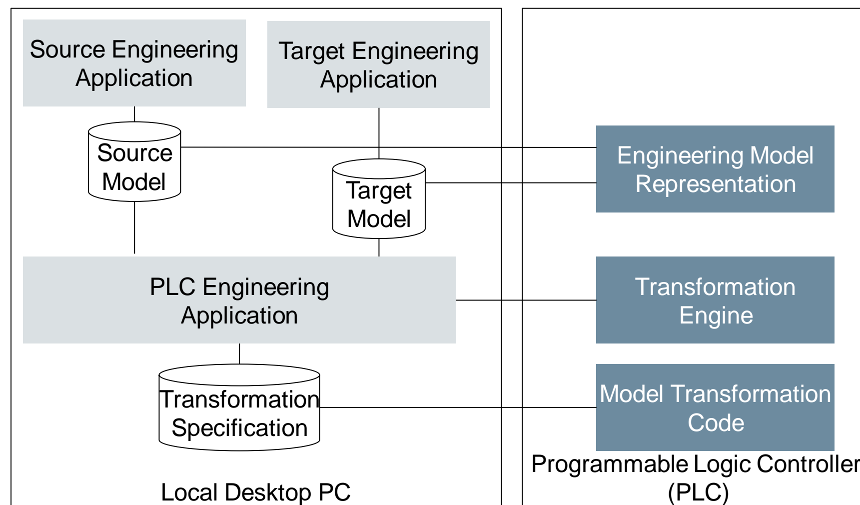

Figure 41: Real-Time Model Transformations

The key difference between a PLC and a general purpose personal computer (PC) is the connection of the PLC to the physical world by sensors and actuators (see Figure 42). By its sensors, the PLC can receive information about its environment. The actuators connected to a PLC control the physical world. In general a PLC program can't be easily restarted because the physical state of the environment must be reversed and hazard for people and equipment must be avoided. Therefore, engineering model transformations must be executed as part of the controller program with respect to the current operating state of the machine.

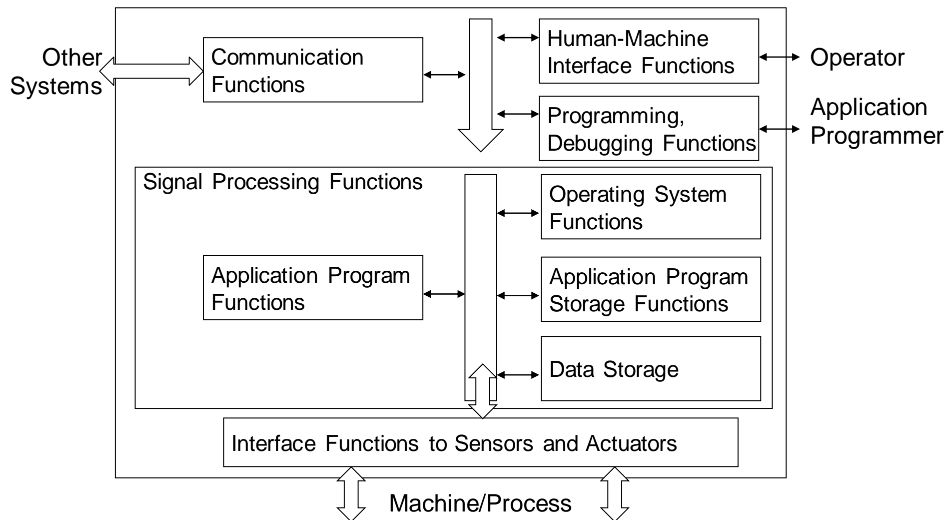

Figure 42: Basic functional structure of a PLC-system [In03a]

The platform specific model of a PLC is the software model described by IEC 61131-3 [In03b]. IEC 61131-3 defines the programming languages which can be used to create the application program shown in Figure 42. The structure of an application program is programming language independent. The application program uses program organization units (POU) and data types (see Figure 43). These elements, POU and data type can be either provided by the application program itself or by the operating system of the PLC, e.g. as built-in elements or as add-on packages from the PLC vendor.

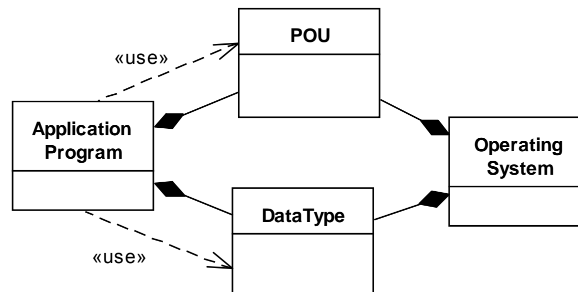

Figure 43: PLC Application Program

The program organization units (POU) represent the executable parts of an application program. IEC 61131-3, 2nd edition, defines 3 different types of program organization units: program, function block, and function (see Figure 44). Despite its similar name, the program POU is does not represent the application program, but is part of an application program on the same level as a function block POU and a function POU. In

contrast to function block POUs and function POUs, a program POU can be assigned to tasks within the PLC. A POU can be called from another POU. This call may include input, output, or input-output parameters. A function is stateless: it does not keep information between subsequent function calls. Function blocks include local data structures and can, therefore, keep state information between subsequent calls. Finally, programs represent the top level POU. They can be assigned to the execution system of a PLC and provide an image of the I/O connections of the PLC.

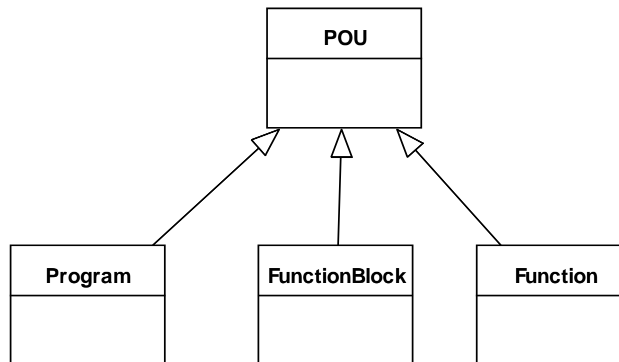

Figure 44: Program Organization Units (POU)

Data types defined by IEC 61131-3 include elementary data types (e.g. Boolean, integer), generic data types which are type compatible to multiple elementary data types (e.g. ANY_NUM for all real and integer data types), and user defined data types (see Figure 45).

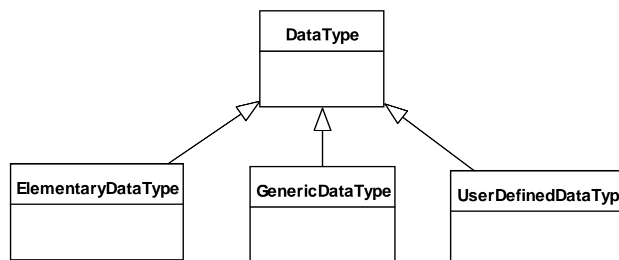

Figure 45: Data Type

Program organization units together with user defined data types are the modeling elements provided by IEC 61131-3 which can be used to describe automation applications. Many PLC vendors provide additional elements like system specific functions, function blocks, and user defined data types as part of their operating system.

These elements can be used together with the IEC 61131-3 model elements to build an application model.

The example model used by this thesis for the PLC configuration of the labeling device application scenario as implemented by a Siemens SIMOTION PLC [Si08a] is shown in Figure 46. SIMOTION provides "Technology Object" and "Positioning Axis" as user defined data types by its operating system for the control of drives. The application program adds the "Label Device Control" function block and the program "Labeling Machine Operation" as part of the software model. The user defined data type "Label Feeder" is part of the electrical engineering model of the labeling device.

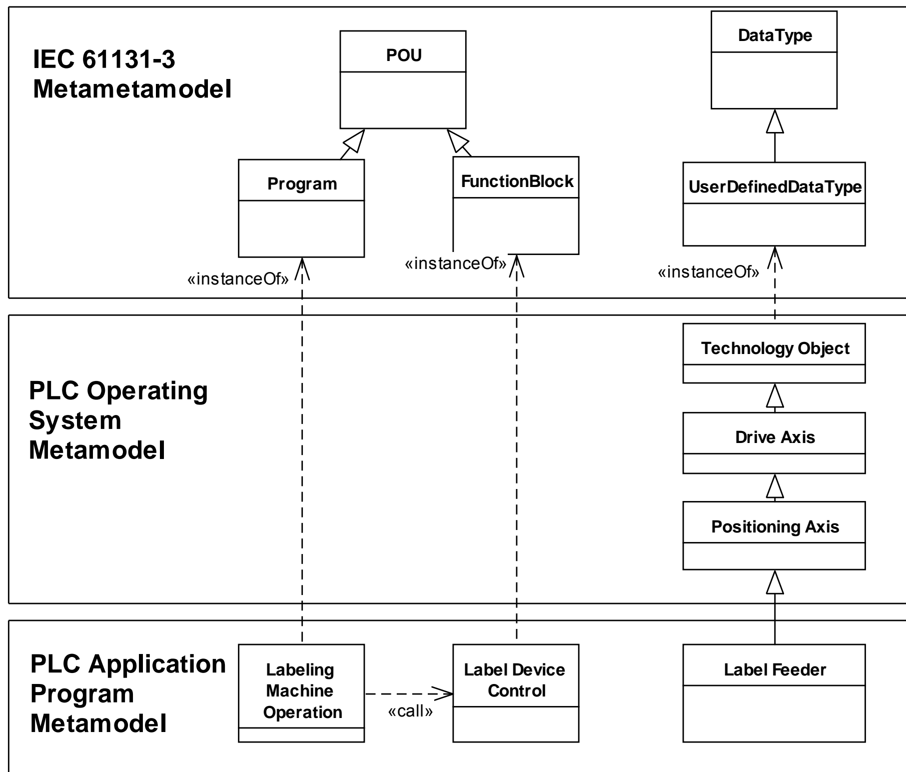

Figure 46: PLC model example: Labeling Device

4.3.1 Rule Language

The IEC 61131-3 [In03b] standard does not include a language for model transformations but includes the high-level programming language structured text (ST) as a general purpose programming language. Therefore, a platform specific rule language was implemented as part of this thesis based on the block oriented programming model of IEC 61131-3 controllers.

The transformation rules are implemented as "transformation rule" function blocks with the programmable logic controller (see Figure 47). Within the PLC controller program, the transformation rule function blocks are executed by a rule execution program as part of the automation program of a production machine. The execution logic of a transformation rule is implemented as structured text (ST) imperative code, which matches a source pattern and builds target elements.

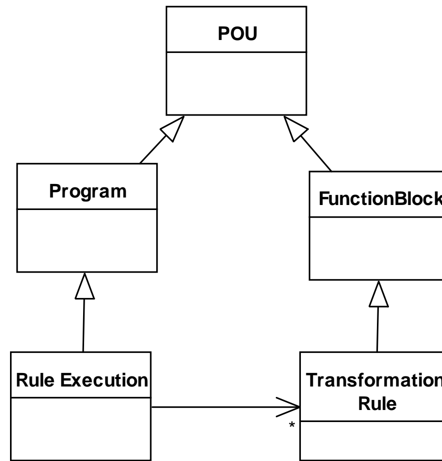

Figure 47: Transformation Rule Function Blocks

In general, model transformation language rules use a typed object as the source context of a transformation rules. Within rule matching, a transformation rule matches objects of the rule context type as well as all objects with an ancestor type within a generalization relationship. The generalization relationship implies that a descendent element can be used wherever an ancestor element is defined [RJB05]. For example, within the model of the labeling machine shown in Figure 46, a rule with a drive axis type for the source context also matches objects with positioning axis type and label feeder type.

The current 2nd edition of IEC 61131-3 [In03b] used by most programmable logic controllers does not include object oriented features like classes and generalization nor references to objects. These features are part of the future 3rd edition of IEC 61131-3 [In12]. Therefore, the IEC 61131-3 model transformation engine implemented as part of

this thesis takes account of generalization relationships by a special implementation of the transformation rule function blocks.

A transformation rule is not implemented by a single function block, but consists of multiple function blocks. Each of these function blocks handles the rule matching for an object type, which is a descendant of the source context object type of a transformation rule definition. For example, a transformation rule with a source context object type of drive axis would be mapped to three transformation rule function blocks according to the generalization hierarchy shown in Figure 46: a function block matching drive axis objects, a function block matching positioning objects, and a function block matching label feeder objects. These function blocks duplicate nearly the same code for each object type. This is acceptable for model transformation rules generated from a platform independent model transformation specification (PIM-MT) as introduced by this thesis. Moreover, the models of many production machines only include a small number of generalization relationships. In the future, with the new features of IEC 61131-3 3rd edition, this code duplication can be avoided by using classes and references.

4.3.2 System Model

The key elements of the system model used by programmable logic controllers (PLC) based on IEC 61131-3 [In03b] were introduced in Figure 44 and Figure 45: function blocks and user defined data types. IEC 61131-3 does not define a model or methods for the introspection of the system model of a PLC. Therefore, an introspection model was implemented as part of this thesis which allows for information access to the elements of the engineering models implemented by the programmable logic controllers. The introspection model includes a function block for each element of the PLC system (see Figure 42) which shall be accessible by model transformation. As shown in Figure 48, function blocks within the introspection model represent the drive objects (like the backLabel and frontLabel drive objects of the labeling machine), technology objects (like the labeling positioning axes), and the application function blocks (like the labeling control function blocks). Depending on the features of the PLC engineering system, the code of introspection function blocks can be automatically generated (e.g. by scripting functions of the PLC engineering systems) or must be manually created by the PLC programmer as part of the PLC software development.

The engineering model elements of the PLC exposed by the introspection model are connected to the physical world by the sensors and actors. Querying the status of engineering model elements and setting the parameters of engineering models often requires the execution of communication procedures for the interaction with the physical devices. Therefore, the introspection model is implemented as function blocks, which can be executed as part of the model transformation process, and not only as simple data structures.

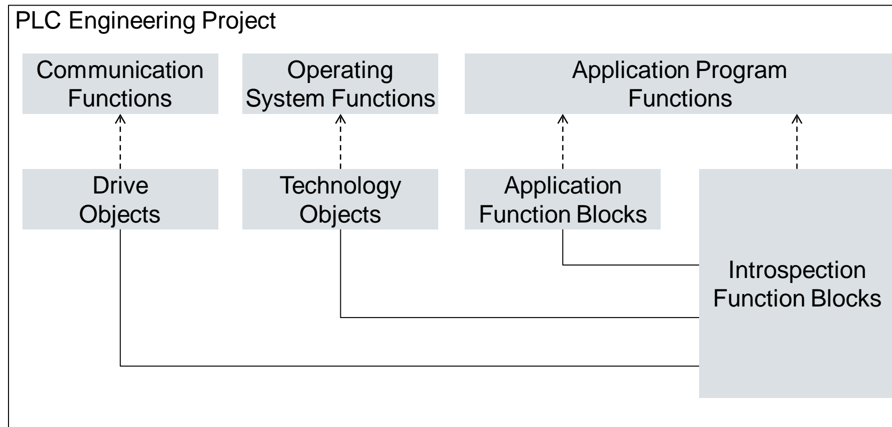

Figure 48: Building the Introspection Model

4.3.3 Pattern Language

Programmable logic controllers (PLC) based on the IEC 61131-3 standard [In03b] do not include a specific programming language for pattern matching. Instead, source patterns and target patterns are evaluated by the imperative programming structured text (ST), which is defined by the IEC 61131-3 standard (see Figure 50).

The source pattern is evaluated by iterating through the elements of the introspection model introduced in Section 4.3.2. As shown in Figure 49, each introspection function block at least holds the name and the active status of a PLC element as part of the meta-information data structure. In addition to the common meta-information, each introspection function block type holds type specific information as references to other objects (driveObject attribute of DriveControlUnit in Figure 49) or the hardware address (address attribute of DriveObject in Figure 49).

The attributes of the source engineering model elements are compared by user defined functions with the filters defined by the source pattern. Special attention must be paid to the selection of the object type of the source context element. The transformation function blocks do not include a query for the object type defined by the source pattern. Instead, a separate transformation function block is used for the object type and all of its super types, which shall be matched by the source pattern. This type specific separation is required, because the 2nd edition of IEC 61131-3 implemented by most controller does not support polymorphic references to function blocks.

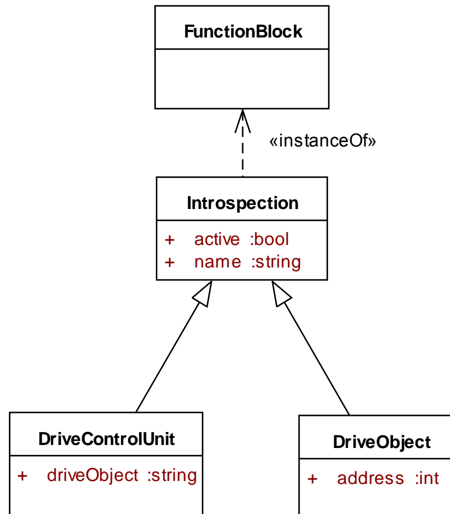

Figure 49: Meta Information about IEC 61131-3 Elements

Elements in the target engineering model are activated by factory functions according to the target pattern. IEC 61131-3 does not define the creation of dynamic instances of functions blocks in favor of predictable memory layout and runtime execution behavior of PLC systems. Therefore, the maximum number of instances of function blocks or user defined data types, which shall be handled by an engineering model, must be statically configured by the PLC engineering system. Within this maximum number of instances, each preconfigured instance might play different roles, e.g. an instance of a function block may control different devices reconfigurable at runtime. Nevertheless, the type and the number of preconfigured elements cannot be modified.

The implementation of the construction of target engineering model elements presented by this thesis assumes that the engineering model element instances required by all valid machine configurations are already present within the PLC but deactivated. The creation operation for an element activates the already available instance. The target pattern example in the lower part of Figure 50 shows the use of the factory function "createLabelDeviceControl" to construct an element of type "LabelDeviceControl" in the target model. Bindings of attributes are assigned by structured text assignments to the input variables of the function blocks (e.g. the "address" in Figure 50). Finally, the function block is executed to transfer the data to the external devices if necessary.

```
VAR_GLOBAL
    driveObjects : ARRAY[1..doNum] OF DriveObject;
END_VAR

// do2toRule for DriveObject type
FUNCTION_BLOCK do2toRule_DriveObject
    VAR_OUTPUT
        modelCompleted : BOOL;
    END_VAR

    // source pattern
    // name filter match
    IF nameCheck(driveObjects[iterator].info.name,
                 'LabelFeeder') THEN
        sourceMatch:=TRUE;
    END_IF;

    // target pattern
    IF sourceMatch THEN
     // create target element
     targetIdentifier:=fbIdentifier(iterator);
     targetIterator:=createLabelDeviceControl(
                                targetIdentifier);
     IF (targetIterator>0) THEN
        // set additional attributes
        labelDeviceControlBlocks[targetIterator].address
          :=driveObjects[iterator].address;
        labelDeviceControlBlocks[targetIterator]()
     END_IF;

    END_IF;

    iterator:=iterator+1;
    IF iterator>_lastIndexOf(in:=driveObjects) THEN
        iterator:=1;
        stateModelCompleted:=TRUE;
    END_IF;

    modelCompleted:=stateModelCompleted;

END_FUNCTION_BLOCK
```

Figure 50: Transformation Function Block Example

4.3.4 Inter-Rule Execution Control

Model transformation rules are executed in a programmable logic controller (PLC) by calls to the rule function blocks from a program (see Figure 51). In IEC 61131-3, a program is a program organization unit (POU), which can be called by the task system of a PLC. The implementation of a PLC task system is vendor specific. For example, the Siemens SIMATIC controllers [Si11b] implement a task system with cyclic execution of programs (called organization blocks for SIMATIC controllers) while the Siemens SIMOTION controllers [Si08a] allow the asynchronous execution of programs. Both task models can be used for the execution of model transformation rules. The cyclic execution model leaves it up to the rule execution engine to ensure that each rule execution cycle does not violate the timing constraints of each execution cycle. The asynchronous execution model schedules the rule transformation engine execution in rotation with other task and relaxes the timing constraints for the rule execution engine. For both scenarios, the rule execution engine must be able to yield control to the PLC execution system and to handle interruptions of the rule execution process by the PLC execution system. Therefore, the execution of model transformation rules in small execution steps was implemented based on IEC 61131-3 as part of this thesis. Transformation rules matching a source pattern with a type specification without any subtypes are implemented as a single function block. Transformation rules matching a source pattern with subtype specifications are split in multiple function blocks for each subtype as described in the previous Section 4.3.3. This allows for lower execution time for each rule execution compared with the execution time of rule specifications, which match more source patterns (e.g. desktop rule specifications).

```
PROGRAM RuleExecution
  VAR
    rule_do2toRule_DriveObject :
                              MT.do2toRule_DriveObject;
    modelCompleted_do2toRule_DriveObject : BOOL:=FALSE;
  END_VAR

  REPEAT
    rule_do2toRule_DriveObject(
        modelCompleted=>
            modelCompleted_do2toRule_DriveObject);
  UNTIL modelCompleted_do2toRule_DriveObject END_REPEAT;

END_PROGRAM
```

Figure 51: Rule Execution Program Example

Beside constraints of the PLC execution system, inter-rule execution must consider constraints of the production process of the production machine. Depending on the process characteristics, the rule execution engine might run continuously or only in specific operating states of the production machine. For example, the machine state model for automatic mode operation of packaging machines [Or06] defines the state

transitions stopped-starting and starting-ready for the initialization of the production machine. For a packaging process, the execution of the rule transformation engine can be part of the starting state of the packaging machine to reflect changes of the machine configuration for a new production batch.

The continuous execution of transformation rules as part of the PLC execution system requires special consideration for the deactivation of engineering model elements. The ATL model transformation rules, used as a PIM-MT within this thesis, only specify the creation of model elements and not the deletion of model elements. Therefore, before the start of execution of model transformation rules, all target engineering model elements are deactivated within a specific operating state of the packaging machine (e.g. as part of the starting state as described in the previous paragraph). Within the continuous execution of the model transformation rules, real-time engineering model elements can delete or deactivate themselves on errors. This reflects inconsistencies of the engineering model which require the further execution of model transformation rules to re-establish engineering model consistency.

Finally, for real-time model transformation engines, the focus of inter-rule execution control is not engineering model specific dependencies between transformation rules but on real-time rule execution constraints. In general, model transformation rules are executed independent of other transformation rules, leading to temporarily inconsistent or invalid engineering models. It is up to the PLC program to handle these temporary inconsistencies and to continue normal operation after engineering model reconciliation.

4.3.5 Modularization

The structured programming model of IEC 61131-3 supports modularization with respect to executable program organization units (POU) and with respect to data structures with user defined data types.

The modularization of program organization units (POU) is used by the IEC 61131-3 model transformation engine implemented as part of this thesis to separate transformation rules by multiple function blocks which can be scheduled within the PLC execution system without the violation of timing constraints. Further modularization, as the aggregation of multiple reusable transformation rules in libraries is not specified by IEC 61131-3 but is part of vendor specific implementations of IEC 61131-3. For example, the library and unit concept of Siemens SIMOTION controllers [Si08a] allows for the specification of production machine independent transformation rules within one library (for example provided by Siemens as the controller provider) and the specification of production machine dependent transformations in another library by the machine provider. The advantage of library concepts for real-time controllers over library concepts of enterprise model transformation engines or desktop model transformation engines is the availability of know-how protection concepts, which protect the intellectual property within the model transformation specification. Model transformation rules implemented as function blocks can be distributed with enabled know-how protection, which hides the implementation code. In contrast to binary distribution, it is possible to reveal the know-how protected code by a key (e.g. a password) for authorized people.

Modularization with respect to data structures is not well supported in the current 2nd edition of IEC 61131-3, since the programming environment only supports aggregation of user defined data types within other user defined data types or within program organization units. This aggregation is used by the implementation of the IEC 61131-3 introspection model of this thesis to share meta-information about IEC 61131-3 elements. Further modularization concepts like inheritance and polymorphism are part of the 3rd edition of IEC 61131-3, but not implemented by most programmable logic controllers until now.

4.3.6 Related Work

The real-time model transformation engine for IEC 61131-3 controllers was developed as a new model transformation engine as part of this thesis. Other model transformation engine implementations on real-time controllers are rarely available. The ACPLT process control system[20] (Aachener ProzessLeitTechnik) provides an implementation of a rule engine based on the IEC 61131 standard. The object management system ACPLT/OV provides introspection and reflection features for the metamodel implemented by ACPLT as required by model transformation engines. The ACPLT/RE rule system [KQE11] uses the ACPLT/OV object management system for the specification of engineering rules for the reconciliation of automation systems. Preparative workings for this thesis [SK12] showed the usage of ACPLT/RE as a platform specific model transformation engine. ACPLT/RE cannot be executed on arbitrary IEC 61131 compatible automation controllers but only on the ACPLT/OV system. The IEC 61131 model transformation engine presented by this thesis can be executed on any IEC 61131 real time controller.

The usage of IEC61131 in model driven environments is part of several workings, for example the generation of automation hardware and software configuration [Ma08], the automation software and simulation models of machine tools [ZP08], or the usage of UML (Unified Modeling Language) and SysML (Systems Modeling Language) as an abstract specification of IEC 61131 systems [FT11]. They consider the generation of automation code from engineering models outside the programmable logic controller but not the integration of the engineering models and the model transformation engine into the real time controller as described by this thesis.

Object orientated programming methods can be used within current IEC 61131 programmable logic controllers either with coding conventions as described by [Ho12] for Siemens SIMATIC PLC or with vendor specific extensions as within CoDeSys [VW07].

4.3.7 Summary

Engineering model representations and model transformation engines are not covered yet by the IEC 61131 standard which describes the structure of real-time programmable logic controllers. The main challenges for the real-time model transformation engine implemented as part of this thesis on top of the IEC 61131 standard were the missing

[20] http://www.plt.rwth-aachen.de/acplt-technologien/

introspection and reflection for IEC 61131 programming elements to explore the PLC system, the lack of dynamic instances for IEC 61131 programming elements to create new model element instances, and the vendor specific access methods to PLC operating system elements like technology objects.

The implementation of a real-time model transformation engine is a new concept created as part of the work of this thesis. It uses IEC 61131 programming elements and coding conventions for the engineering model representation and the model transformation rules. The IEC 61131 implementation of the model transformation engine cannot check, if all elements required for the execution of a transformation rule are correctly coded (e.g. the definition of rule patterns, the integration of the rule into the rule system, and the integration of rule related model element in the engineering model representation). Therefore, the generation of the IEC 61131 from a platform independent rule specification as described in the next section is superior to manually coding the model transformation engine.

PSM-MT feature	**IEC 61131-3**
Rule Language	rule function blocks
System Model	generated/manually programmed IEC 61131 representation
Pattern Language	custom functions
Inter-Rule execution control	controlled by machine state machine
Modularization	IEC 61131 program organization units and user defined data types

Figure 52: IEC 61131-3 PSM-MT features

4.4 PSM-MT summary

For industrial usage, standardized and approved solutions are required for implementation of model transformation engines. The three platform specific model transformation systems (PSM-MT) ATL, XSLT and IEC 61131-3 are not only selected as an application example but as the standard technology representing the three execution environments desktop model transformations, enterprise model transformations, and real-time model transformations. For desktop model transformation languages, no commonly used standardized languages exist until now. Therefore, the ATL transformation was used as a representative for the characteristics of declarative model transformation languages because of the maturity of the ATL implementation and its tool support. Within enterprise model transformation systems, XSLT [Wo07] was chosen because of its usage on middleware servers connecting engineering systems while SQL [IS11] is only used for backend database servers. Real-time controllers are based on IEC 61131 [In03a]. Beside the introduction of the three platform specific model transformation environment required for the engineering of production machine, the objective of this section was the definition of common model transformation concepts available on all model transformation platforms for usage within platform independent to platform specific model transformation transformations.

The rule languages used for platform specific engineering model transformations are based on simple declarations of the mapping of a source pattern to a target pattern. These rule specifications can be easily expressed by all three platform specific model transformation engines considered by this thesis.

The system model of ATL, the Ecore metamodel, provides fewer features for the definition of element constraints than the XSLT system model, the XML schema definitions. The system model elements of IEC 61131 controllers are not explicitly modeled but are implicitly part of the operating system or the user program. Therefore, the usage of system model elements of Ecore and XSLT was restricted to prepare the ground for a platform independent system model. IEC 61131 does not explicitly define a system model. Therefore, the system model of IEC 61131 was realized within this thesis by user defined types and function blocks, which allow for the introspection of IEC 61131 elements for model transformations. This system model within real-time controllers can be created manually or can be generated automatically within the engineering system of IEC 61131 controllers.

The pattern languages used by ATL, XSLT, and IEC 61131-3 are very different in their language structure and their language features. Therefore, complex queries and complex object construction are encapsulated in user defined functions, which are available on all three model transformation platforms. This allows for the platform specific optimization of the patterns used within the engineering of a production machine, while the pattern specification remains platform independent.

Inter-rule execution control is based on different specification concepts in the three platform specific transformation engines. ATL tries to hide the inter-rule execution

control from the user and uses a two pass rule execution algorithm, which handles element creation and element references in two different steps. This hidden inter-rule execution control does not consider rule execution requirements of the target system but only considers the data structure of the engineering model. XSLT allow partly control about rule execution by explicit selection of the parts of the engineering model, which should be considered for rule application. Moreover, the middleware used in an enterprise server also has control about rule execution by selecting the parts of the engineering model which are handled for example as part of the message within a transformation pipeline. This selection might be based for example on element locking for multi-user systems or on change events within the engineering system. IEC 61131-3 model transformations are not much influenced by the model structure but by the real-time execution constraints, which require a small execution granularity and time limits for the execution of a single rule.

Finally, modularization covers multiple aspects: the modularization of the rule specifications, the modularization of the rule execution, and the modularization of the system model. Within the model transformation platforms, only the modularization of the rule specifications is included. All three model transformation platforms support the modularization on the level of source files. Multiple files can be used to group the definition of rules used by a main module. This allows for the reuse of rules as part of libraries. Know-how protection of rule specification libraries is only part of IEC 61131 controllers as a vendor specific extension. The other model transformation systems do not provide specific support for know-how protection.

5 Platform Independent Model Transformation Language

Within the application example, the bottle labeling machine, the model transformations executed for the reconciliation of the engineering models of a machine configuration are executed on different platform specific model transformation engines as described in Section 4 depending on the machine requirements. The same set of transformation rules shall be executed on a desktop model transformation engine, on an enterprise model transformation engine, and on a real-time programmable logic controller.

The model driven architecture (MDA) [MM03] describes the generation of platform specific models (PSM) from more abstract platform independent models (PIM). The platform independent model hides implementation details of the specific implementation platforms and allows for the specification of a system independently of the platform that supports it. This enables the reuse of an implementation on multiple platforms. A model transformation maps the platform independent model (PIM) to a platform specific model (PSM), which provides an implementation of the PIM on a specific execution environment. MDA considers model transformations as a technology used outside the platforms, which shall be transformed. Therefore, this thesis proposes the extension of the model driven architecture (MDA) approach to the transformation of platform independent model transformations (PIM-MT) to platform specific model transformations (PSM-MT) (see Figure 53). This enables the usage of the modeling and transformation environment together with the system to be modeled and transformed on the same platform. The transformation and the transformed system are no longer separated but available on a common platform. This allows for the usage of reconfigurable systems on multiple platforms as required for example for machines like the bottle labeling machine of the application example in Section 2.

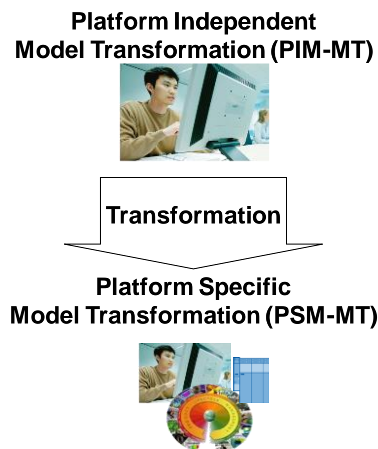

Figure 53: Model Driven Architecture PIM to PSM transformation

According to the MDA Guide [MM03], the application of the model driven architecture includes multiple steps:

"MDA provides an approach for, and enables tools to be provided for:

- specifying a system independently of the platform that supports it,
- specifying platforms,
- choosing a particular platform for the system, and
- transforming the system specification into one for a particular platform.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns." [MM03]

The specification of the platforms, which can be chosen as a particular platform for the operation of a production machine, was described in Section 4 (Figure 54 shows the three platform specific model transformations for desktop, enterprise, and runtime execution). This Section describes the system independent specification of model transformations used as a platform independent model PIM-MT (upper part of Figure 54). The transformation between the PIM-MT and PSM-MT is described in detail in Section 6.

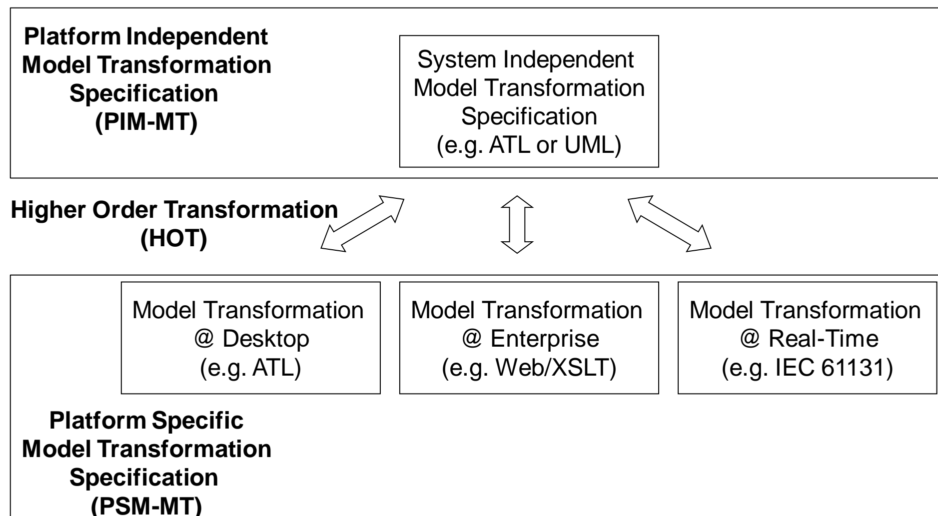

Figure 54: PIM-MT/PSM-MT Platform Scope

The new approach presented in this thesis considers a platform independent specification of model transformations not as a specification, which can be executed anywhere, but as a specification, which can be transformed to a platform specific specification executable on different target systems. For the definition of a platform independent transformation specification, three different approaches can be considered: the specification of a completely new transformation language, the specialization of an existing general purpose language (e.g. the unified modeling language (UML) [IS12a]), or the generalization of an existing model transformation language. The design of a new

transformation language was not considered as part of this thesis since the availability of a stable implementation of the model transformation specification is a key requirement for industrial usage. The specialization of UML was considered within the preparative work of thesis. This option was discarded due to the complexity of the UML language together with the missing tool support for transformations from the UML metamodel to other models. Therefore, the generalization of an existing model transformation language was chosen as a platform independent specification of model transformations as described in the next section.

For the work of this thesis, the ATL transformation language [Jo06] was tailored for the usage as a platform independent transformation language. ATL was chosen for multiple technical reasons. First of all, ATL provides a mature implementation based on the Eclipse platform[21] with commercial technical support (e.g. from Obeo[22]). ATL uses a textual representation of model transformation rules, which can be easily handled by PLM or version control systems. A parser, which translates the textual representation in an Ecore model instance, is provided as part of the ATL implementation (the detailed Ecore model is described by [Ti09]) This Ecore model can be used by the PIM-MT to PSM-MT transformation according to the model driven approach chosen by this thesis (see Section 3.2).

ATL is not a platform independent transformation language. Within this thesis, the syntax and parts of the ATL semantics are used for the platform independent specification of model transformations. With this approach, the specifications and the tooling of the ATL language can be reused as a platform independent modeling language. Generalization of the ATL means that concepts of the ATL language, which cannot be used on multiple platforms, are omitted. In the following, the syntax and semantics of ATL, as used for platform independent model transformations are described.

---

[21] http://eclipse.org/atl/
[22] http://www.obeo.fr/

5.1 Rule Language

The ATL rule language includes three different rule specifications: matched rules, lazy matched rules, and called rules. Called rules can be used within ATL imperative code. Lazy rules allow for the modularization of ATL rules. The platform independent transformation specification defined by this thesis only uses the matched rules for the declarative rule specification. The original ATL syntax specification for matched rules is shown in Figure 55, the generalized syntax used for platform independent rule specification is shown in Figure 56.

```
rule rule_name {
  from
    in_var : in_type [in model_name]? [(
        condition
    )]?
  [using {
    var1 : var_type1 = init_exp1;
      ...
    varn : var_typen = init_expn;
  }]?
  to
    out_var1 : out_type1 [in model_name]? (
      bindings1
    ),
    out_var2 : distinct out_type2
                 foreach(e in collection)(
      bindings2
    ),
    ...
    out_varn : out_typen [in model_name]? (
      bindingsn
    )
  [do {
    statements
  }]?
}
```

Figure 55: Original ATL Matched Rule Syntax[23]

The "from" section of the ATL rule defines a source pattern for matching a single object. The "to" section of the ATL rule defines multiple target patterns for the creation of objects. Both patterns require objects with type definitions. ATL rules benefit from models with a detailed type system. Otherwise, the specification of object matching and construction is more difficult, because complex matching conditions, which are harder to

[23] http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#Matched_Rules

specify and maintain, are required besides the type matching. Therefore, platform specific model transformation should add a layer for the mapping to a detailed type system if they originally provide only a simple type system to ease the specification of platform independent model transformation rules.

The main elements omitted from the original rule syntax are the imperative statement part at the end of the rule specification (the "do" block) and the local variables section (the "using" block), because they are difficult to handle by a higher order transformation. The imperative statement part can be used for evaluations across rules, e.g. adding model elements to global variables. The using block allows the reuse of expression values (e.g. a name string) at multiple places within a rule. The iterative target pattern element defined by "distinct … foreach" is excluded because it is deprecated in ATL. Finally, the platform independent transformation specification is restricted to a single input model and a single output model. Therefore, the specification of the referenced model by the "in" keyword is not used.

```
rule rule_name {
      from
            in_var : in_type [(
                  condition
            )]?
      to
            out_var1 : out_type1 (
                  bindings1
            ),
            out_var2 : out_type2 (
                  bindings2
            ),
            ...
            out_varn : out_typen (
                  bindingsn
            )
}
```

Figure 56: Generalized ATL Matched Rule Syntax

5.2 System Model

The ATL model transformation language references input and output metamodels based on Ecore [St09] meta-metamodel definitions (see Figure 57). The engineering metamodels of production machines only use a small subset of the Ecore meta-metamodel features as specified in Section 3.5: EClass, EAttribute, and eSuperTypes. Instead of EReference, the engineering metamodels use attributes with reference designators as required by mechatronic models.

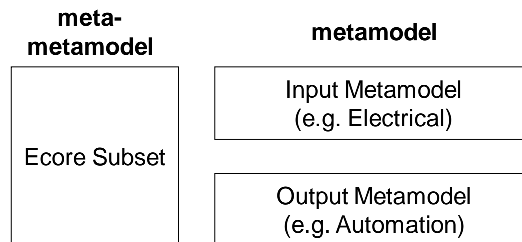

Figure 57: Platform Independent System Model

The input and output metamodels could be defined manually on the level of platform independent specifications. But in general, a platform specific engineering system already provides existing metamodels, for example PLMXML schema definitions for enterprise model transformations. Therefore, a metamodel transformation from the platform specific metamodel to the platform independent metamodel is required as shown in Figure 58. This is an extension to the model driven architecture (MDA) point of view, which assumes in general transformations from platform independent models to platform specific models as shown in Figure 53.

Within the work of this thesis, it is assumed that it is possible to specify a metamodel for every platform specific model. The development of such a specification is required if the platform specific model transformation does not provide a predefined metamodel specification. For example, the IEC 61131-3 standard used for the real-time model transformation engine (see Section 4.3) did not provide a metamodel specification. Therefore, a metamodel based on the IEC 61131-3 programming languages specification was created as a PSM-MM as part of the work of this thesis. The answer to the question, if it is possible to specify a metamodel for every engineering model, is outside the work of this thesis.

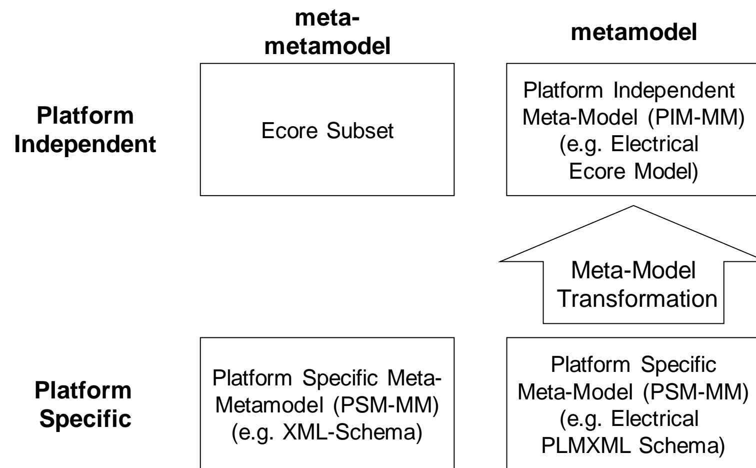

Figure 58: PSM-MM to PIM-MM Transformation

On the level of platform independent model specification, the metamodel specification is useful for the verification of the platform independent model transformation specification:

- Static type checks can validate types and attributes used by the model transformation rules. The ATL model transformation language used as a platform independent model transformation specification by this thesis does not provide static checks. The ATL transformation engine checks the metamodel at runtime using the reflective Ecore interfaces. Static checks could be implemented for ATL by a transformation from the ATL model transformation specification to a problem model as described by [AT05]. These checks have not been implemented as part of this thesis.
- Code completion and code proposals based on the referenced metamodels help for the specification of platform independent model transformation rules. The ATL editing environment supports code completion for types and attributes specified by the referenced metamodels.
- Test suites can be executed on the level of platform independent model transformation specifications. This is an advantage of using a model transformation engine like ATL as a platform independent model transformation specification instead of a common data exchange format or abstract specification without execution engine. The metamodel specification can be combined with test models on the platform independent level to test the correctness of model transformation rules.

The PSM-MM to PIM-MM transformation generates multiple platform independent metamodel specifications, which can be referenced by the same platform independent metamodel specification. The multiple platform independent metamodels targeted for the same model transformation specification can be handled with three different options:

- The metamodels can be merged to a single platform independent metamodel, which holds a union of all types created from the platform specific metamodels. In this case, not all platform independent transformation rules can be executed on all specific transformation platforms.
- A single platform independent metamodel can be created as an intersection of the types created from the platform specific metamodels. In this case, the platform independent model transformation specification only includes rules executable on all specific transformation platforms. Platform specific transformation rules must be specified on the specific transformation rules and merged with the rules generated from the platform independent transformation rule specification.
- The metamodels can stay separately and be selected within modular platform independent rule specifications as part of ATL launch configurations (see Section 4.1.5). This approach was chosen within the implementation of this thesis, because the ATL modularization allows for flexible selection of the appropriate platform independent transformation rules specifically for each PIM-MT to PSM-MT transformation without a-priori decision about merging or intersecting metamodels.

5.3 Pattern Language

The rule language and the Ecore system model used by the ATL platform independent transformation specification are based on a simple, abstract model which is well suited to be handled by a PIM-MT to PSM-MT model transformation. The pattern language of model transformation specifications is mostly a complex language to support the specification of complex source or target patterns as required by engineering model transformations. These pattern languages are hard to handle by a PIM-MT to PSM-MT model transformation because of the complex abstract syntax trees representing these languages. The translation of a complex pattern language would be better solved by a compiler. This would break the MDA concept of model transformations and prevent users to create the PIM-MT to PSM-MT transformation with the same knowledge which is required to specify the platform independent model transformation specification.

Therefore, this thesis proposes to restrict the definition of source and target patterns to the usage of user defined functions and string literals. User defined functions provide a level of abstraction suitable for the usage of model transformations for the transformation of PIM-MT patterns to PSM-MT patterns. With user defined functions, the same patterns can be realized as with the usage of a complex platform independent pattern language for the cost of the reimplementation of each user defined function on the platform specific level by the means of the platform specific pattern language. An example for the usage of user defined functions is shown in Figure 59. The upper part shows the rule "do2to" with expressions used in the rule definition: the source pattern uses the filter "s.name.startswith('backLabel')" and the target pattern the binding "'fbrd_'+s.name". In the lower part, the rule "do2to" is shown with the user defined function "thisModule.nameCheck(s.name, 'backLabel')" replacing the filter "s.name.startswith('backLabel')" and the user defined function "thisModule.RDLabelDeviceControlName(s.name)" replacing the binding "'fbrd_'+s.name". The user defined functions "nameCheck" and "RDLabelDeviceControlName" provide an encapsulation as well as an abstraction of the meaning of the expression.

Beside user defined functions, string literals can be used to represent constant values. Other literals like integer number or floating numbers are not included in the implementation provided by this thesis since strings are available on all platform specific model transformation system. With the help of user defined functions, string literals can be converted to other data types if required.

```
-- Usage of Expressions
rule do2to
{
	from
		s: MMELECTRICAL!DriveObject
		(
			s.name.startswith('backLabel')
		)
	to
		u: MMAUTOMATION!LabelDeviceControl
		(
			name <- 'fbrd_'+s.name
			, address <- s.address
			, labelformat <- 'f203'
		)
}
```

```
-- Usage of User Defined Functions
helper def : nameCheck
	(value : String, substring : String)
	: Boolean =
	value.startsWith(substring);
helper def : RDLabelDeviceControl
	(driveObject :      MMELECTRICAL!DriveObject)
	: String =
	'fbrd_'+driveObject.name;
rule do2to
{
	from
		s: MMELECTRICAL!DriveObject
		(
			thisModule.nameCheck(s.name, 'backLabel')
		)
	to
		u: MMAUTOMATION!LabelDeviceControl
		(
			name <-
		thisModule.RDLabelDeviceControlName(s.name)
			, address <- s.address
			, labelformat <- 'f203'
		)
}
```

Figure 59: Pattern language with Expressions vs. User Defined Functions

Within target patterns, the ATL pattern language supports a special construct for object references: object references in the target model can be assigned to references from the source model. The code snippet in Figure 60 shows a rule for transformation of a drive control unit to a controller function block named "cu2control", which assigns a reference to a labeling device with the binding "labelingDeviceRef <- s.driveObjectRef". This assignment is not valid, because the variable "s" refers to an element from the source model. ATL handles this assignment internally with traceability links: the reference is initialized with the target model element, which is created by the default rule defined for the source model object referenced by "s.driveObjectRef". ATL creates this reference as a strong reference based on the EReference element from the Ecore metamodel. A strong reference invalidates the complete model if the reference is violated (e.g. the referenced element is not available). EReference elements can reference anonymous objects without visible identifiers.

```
rule do2to
{
	from
		s: MMELECTRICAL!DriveObject
	to
		t: MMAUTOMATION!LabelDeviceControl
		(
			name <- 'fbrd_'+driveObjectName
		)
}
rule cu2control
{
	from
		s: MMELECTRICAL!DriveControlUnit
	to
		t: MMAUTOMATION!LabelDeviceCuControl
		(
			name <- 'fbrd_'+driveControlUnitName.name
			, labelingDeviceRef <- s.driveObjectRef
		)
}
```

Figure 60: Object Reference Handling by ATL Traceability Links

For the platform independent specification of engineering model transformations, weak references are used in ATL instead of strong references. Weak references are preferable for engineering model transformations for the following reasons:

- Weak references can be used if the platform specific model does not support strong references.
- No a posteriori resolution of traceability links is required for the platform specific model transformation. The platform specific model might not support the creation and storage of a traceability model. Each rule can be executed atomically, because the reference to another object can be calculated by an identity function even if an object is not created yet. This is not possible with anonymous references as used by most model transformation languages. The additional execution time for the resolution of the traceability links is not required.
- Weak references allow for the transformation of models with invalid references between model elements. For example in enterprise transformation systems, parts of engineering model might be locked within configuration management or the enterprise server might tailor big models to smaller transformation units.

Within the work of this thesis, weak references are implemented on the base of reference designators in string attributes of the engineering model. In contrast to anonymous objects, which can be referenced within Ecore models, engineering models for production rules usually identify elements by visible reference designators. Reference designators are for communication and identification within the production machine lifecycle with engineering, commissioning, and production. Reference designators can be defined internally by company standards (e.g. naming standards) or by common standards like [IS09]. Compared with strong references, a disadvantage of weak references is the lack of type checking for the referenced elements due to the string representation of references.

For the use within engineering model transformations, reference designators shall be created within user defined functions (e.g. the function "RDLabelDeviceCuControl" creates the reference designator for LabelDeviceCuControl in Figure 61). These user defined functions can be used within object creation (e.g. the assignment of the object attribute "name <- thisModule.RDLabelDeviceControl(s.name)" for the creation of "DriveObject" in rule "do2to"). The same functions can be used to create the references between objects (e.g. the reference assignment "labelingDevice <- thisModule.RDLabelDeviceControl(s.driveObjectRef.name)" in rule "cu2control"). With the help of the reference designator creation functions, references can be already created, even if the referenced object does not exist yet. On the other side, the usage of the wrong reference designator creation function would not be detected on the level of platform independent transformation rule specification. For one to many references, the reference designator creation function can be encapsulated in a user defined function, which resolves an array of reference designators. These user defined functions can be defined without influence on the transformation from the platform independent transformation model to the platform specific transformation model.

```
helper def : RDLabelDeviceCuControl(driveControlUnitName :
String) : String = 'fbrd_'+driveControlUnitName;
helper def : RDLabelDeviceControl(driveObjectName : String) :
String = 'fbrd_'+driveObjectName;

rule do2to
{
	from
		s: MMELECTRICAL!DriveObject
	to
		u: MMAUTOMATION!LabelDeviceControl
		(
			name <-
			thisModule.RDLabelDeviceControl(s.name)
		)
}
rule cu2control
{
	from
		s: MMELECTRICAL!DriveControlUnit
	to
		t: MMAUTOMATION!LabelDeviceCuControl
		(
			name <-
			thisModule.RDLabelDeviceCuControl(s.name)
			, labelingDevice <-
	 thisModule.RDLabelDeviceControl(s.driveObjectRef.name)


		)
}
```

Figure 61: Weak Object References by Reference Designators

5.4 Inter-Rule Execution Control

The ATL model transformation language does not support the definition of the execution order for matched rules used for platform independent specification of engineering model transformations. The transformation rules are executed in the order of their definition within the transformation module. In ATL, rules cannot be executed atomically because of the resolution of object references by the traceability model. Atomically means in this context that each model transformation rule can be executed independently of each other model transformation rule. The ATL transformation executes all transformation rules, keeping track of unresolved references within the

traceability model. After finishing the execution of all model transformation rules, the unresolved references are fixed within the target model to provide the final model transformation result.

For platform independent model transformations, each transformation rule shall be executable independent of the other transformation rules as an atomic model transformation operation to support platform specific model transformation systems with different rule execution algorithms. For example, for enterprise model transformation or real-time model transformations, long running model transformations are hard to handle within the timing restrictions of the execution environment. Therefore, the ATL solution of executing all model transformation rules and resolving references within the complete target model is difficult to use for platform specific model transformation specifications. The usage of weak references as introduced in Section 5.3 solves this problem: references are calculated immediately on the execution of each transformation rule. Therefore, open references need not to be fixed a-posteriori in the complete target model.

A model transformation engine, which executes continuously on existing models, must handle the deletion of target model elements, because the continuous execution of atomic model transformation rules only affects the parts of the target model, which are addressed by the target pattern of a model transformation rule, and not target model elements, which are not required anymore due to a change of the source model. Current model transformation languages like ATL in general only specify the creation of elements but not the deletion of elements. Therefore, the deletion of target model elements must be handled by the platform specific model transformation execution environment independently from the platform independent model transformation specification. The platform specific model transformation execution control can handle element deletion on different levels shown in Figure 62: as part of a platform specific model transformation execution environment: within the engineering models (e.g. as part of state changes of engineered system), within the model elements extracted from the engineering models as part of the model access (e.g. as part of a message queue within an enterprise model transformation environment), or within the platform specific model transformation engine (e.g. as explicit delete operations).

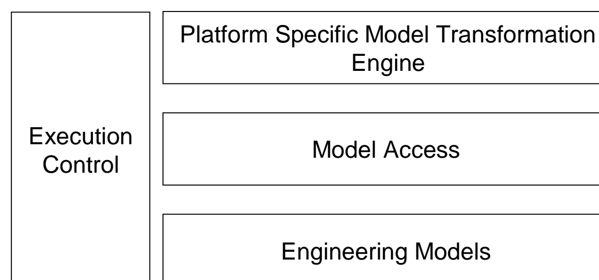

Figure 62: Platform Specific Execution Control

Based on the structure of the platform specific model transformation environments shown in Figure 62, the following strategies have been identified for the handling of target element deletion within the work of this thesis:

- Deleting target elements dependent on a technological model transformation context.
- Deleting target elements dependent on the system state.
- Deleting target elements dependent on deleted source elements.
- Manual specification of target element deletion outside the model transformation specification.
- Deleting target elements dependent on a supermodel, which merges source and target model.

The technological context for the model can be provided by the structure of the engineering model of the production machine. For example, the engineering model of the production machine can be structured according to ISA 88 [Am10] into process cell entities, unit entities, equipment module entities, and control module entities for the physical part (represented by a source engineering model based on an electrical engineering metamodel) and for the control part (represented by a target engineering model based on an automation engineering metamodel). The execution control can iteratively select model entities, prepare the model transformation by deleting the target elements for the selected model entity, and trigger the execution of the model transformation rules, which (re-)create the target model elements required for model reconciliation.

Without consideration of the engineering model structure, the deletion of target elements can be controlled by the system state. For example, the OMAC state machine of a packaging machine [Or06] can trigger an initialization of the source engineering model and the deletion of target model elements for platform specific model transformation executed on a real-time controller.

The option to delete target model elements dependent on deleted source elements is also well suited for real-time controllers or other systems without dynamic object creation. Normally, model transformation rules cannot match deleted source model elements, since deleted elements are no longer available within the source model. For systems without dynamic memory management like real-time controller, model elements are not dynamically created or deleted but a set of preconfigured objects is activated or deactivated. This helps for a defined memory layout without memory overflow problems and for a defined timing behavior. Therefore, deleted model elements can be matched as inactive objects within the memory of a real-time controller. The platform specific model transformation rule can match active elements as well as inactive elements and create or delete target model elements accordingly (which means activation or deactivation of target model elements on real-time controllers).

Besides the handling of target element deletion based on the model structure and system characteristics, a common approach is the manual definition of the execution control

using build specifications like MAKE[24] or ANT[25]. With the knowledge about the structure of engineering models from the design process of a machine, the preparation of engineering models for transformation execution and the execution control of model transformation rules can be defined similar to the build processes of complex software systems.

The last option, deleting target model elements dependent on a supermodel, which merges source and target model, is difficult to implement for engineering model transformations. This approach is implemented in some current model transformation languages like the refinement mode of ATL[26] or the correspondence model of triple graph grammars (TGG)[27]. The refinement mode of ATL assumes that source and target model are combined in a single model, which is usually not possible for engineering models handled within different engineering domains. The TGG correspondence model is a third model beside source and target, which keeps track of the relationship between source and target model elements, supporting the deletion of target model elements, if the source model element no longer exists. The definition and storage of an additional engineering model is hard to handle by different platform specific model transformation systems.

For the work within this thesis, the first two options have been selected for the implementation of the platform specific transformation of the engineering models of a labeling machine. The first option, using a technological context was used for the evaluation of model transformation rules affecting only the labeling unit equipment module of a labeling machine by desktop and enterprise model transformations. The second option, executing the delete operation based on system state, was used for evaluation of engineering model transformations by real-time controllers.

5.5 Modularization

The ATL modularization concept of the separation of the system model from the model transformation modules is used to combine different system models generated from platform specific model transformation engines to a launch configuration for a platform independent model transformation specification as described in Section 5.2.

The second ATL modularization concept, grouping ATL model transformation rules in separate modules, which are used by a main module, is not required for platform independent model transformation specifications, because model transformation rules can be executed independent from each other as atomic transformations. Therefore, platform independent model transformation rules can be split into multiple ATL modules without the requirement to include ATL model transformation rules into a single ATL module.

---

[24] http://www.gnu.org/software/make/
[25] http://ant.apache.org/
[26] http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#ATL_Refining_Mode
[27] http://www.moflon.org/

5.6 Related Work

The reuse of an existing modeling language as a platform independent specification language was already considered for the unified modeling language (UML). The usage of UML for model transformation design was proposed by [Jé05] for a tailored UML subset. The UMLX extension of UML [Wi03] is designed as a specific model transformation engine, which compiles to different target languages like XSLT or Java. Using UML as a platform independent model transformation language introduces a high level of complexity due to many unneeded UML constructs and lacks a tailored toolset for model transformation engineering.

For graph transformations, the graph transformation exchange language GTXL [La04] was specified to exchange transformation specifications of graph transformation-based model transformation tools. Being focused on graph transformation tools and without tool support, GTXL is not suitable as a platform independent model transformation specification.

Missing tool support and high complexity is also the drawback of the "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification" (QVT) [Ob11]. QVT is designed as a model transformation language, which can be implemented by different provider on different implementation platforms. QVT model transformation specifications shall be executable on all QVT implementations, which adhere to the QVT standard. In contrast to the reuse of existing transformation technologies on different platforms like XSLT or IEC 61131 on different platforms as proposed by this thesis, a QVT engine would be an additional component on each platform.

The classification of platform specification model transformation languages in desktop, enterprise, and real-time transformation engines introduced within this thesis is similar to the concept of technical spaces [KBA02]. The application of technical spaces with respect to bridges between modeling frameworks in [BK05] considers technical spaces on the same level of abstraction and not on different levels of abstraction as for the PIM-MT to PSM-MT according to MDA as implemented by this thesis.

The survey about the interoperability of model-to-model transformation of [JK07] states: "language interoperability […] is an ability to execute programs written in one language with the tools designed for another language" [JK07]. This definition is similar to the transformation from a PIM-MT to a PSM-MT. The survey provides mainly comparison of language features to allow for the selection of a language and heuristics for language evaluation but no approach for transformations between languages.

The model transformation of model transformation specification models as used for the PIM-MT to PSM-MT by this thesis is called a higher-order transformation (HOT) for the ATL transformation language by [Ti09]. A higher order transformation does not transform a model into another model but transforms a model transformation model into another model transformation model. Higher order transformations are used to modify the ATL transformation specification itself (e.g. for refactoring ATL transformations [Wi12]) but not for the transformation of an ATL model transformation model to the model transformation model of another model transformation language.

5.7 Summary

Platform independent model transformation specification has been considered until now mainly under the aspect of exchanging model transformation specifications between different model transformation engines and not from the aspect of mapping a platform independent model transformation to platform specific model transformation specifications as an extension to the MDA approach.

Within this section, the first part of the platform independent model transformation specification (PIM-MT) to platform specific model transformation specification (PSM-MT) transformation, the platform independent model transformation specification, has been introduced. Instead of the definition of a new platform independent model transformation specification language, a proven existing model transformation language with a mature toolset, the ATL language, was tailored for the usage as a platform independent. Tailoring covers the identification of language features commonly available on the target platform specific model transformation environments as well as the preparation of the transformation from the PIM-MT to the PSM-MT by a higher-order model transformation as introduced in the next section. The usage of ATL as a platform independent transformation specification does not introduce new language features but only restricts the usage of existing language features.

The declarative part of the ATL rule language can be used as a platform independent rule language with minor modifications.

Major modifications are required for the system model and the pattern language. The system model used by platform independent model specifications has been restricted to the usage of the EClass and EAttribute features, replacing the strong references EReference by weak references based on technological reference designators. Reference designators are created within platform independent model transformation specifications by user defined functions, which are used for object identification, avoiding anonymous objects, as well as for reference definition. User defined functions also play an important role within the platform independent pattern language. The pattern languages of platform specific model transformation languages introduce a high level of complexity and are difficult to generate by a model transformation. Therefore, user defined functions are used as an additional level of abstraction within the platform independent and platform specific pattern language. This allows for the transformation from PIM-MT to PSM-MT by model transformation with still keeping the support of complex pattern definitions.

The platform independent specification of model transformation rules assumes the atomic execution of model transformation rules. Therefore, inter-rule execution control mainly deals with the execution control of the model transformation execution based on target system requirements and the handling of the engineering models (e.g. preparation of target models for the execution of model transformation rules by the deletion of target elements).

Finally, the modularization of platform independent model transformation specification allows for the configuration of platform independent model transformation specifications

with respect to specifics of platform specific model transformation specifications, mainly for the system model used.

The usage of ATL as a platform independent model transformation specification is summarized in Figure 63. The platform independent model transformation specification (PIM-MT) by ATL, which references platform independent metamodels based on the Ecore meta-metamodel, is transformed to platform specific model transformations (PSM-MT) referencing platform specific metamodels. The directed associations between the elements in Figure 63 reflect the navigability between the elements (the transformations have an association to the models handled by the transformation; the models have an association to their metamodels). The next section describes the implementation of the higher order transformation between the PIM-MT and PSM-MT specifications together with the inverse transformation form PSM metamodels to PIM metamodels for the three model transformation platforms desktop, enterprise, and real-time, considered by this thesis.

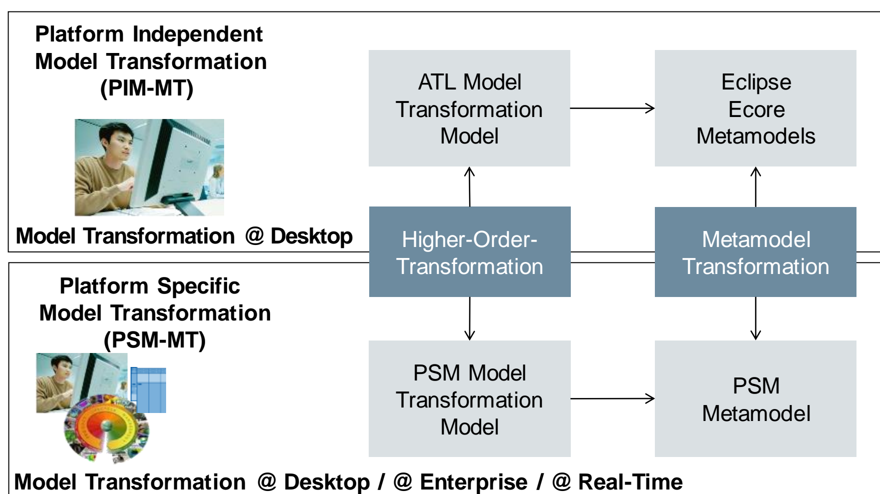

Figure 63: PIM-MT to PSM-MT Transformations

6 PIM-MT to PSM-MT Transformations

This section describes the evaluation of PIM-MT to PSM-MT transformations for desktop, enterprise, and real-time transformations, as implemented by this thesis. As presented in Section 5, the PIM-MT to PSM-MT transformation includes two transformations: the higher-order-transformation from the platform independent model transformation specification to the platform specific model transformation specification and the reverse directed transformation of the platform specific metamodel to the platform independent metamodel.

The PIM-MT to PSM-MT transformations are implemented by the ATL transformation language. ATL is used both for the platform independent specification of model transformations and for the PIM-MT to PSM-MT transformation. Therefore, the ATL know-how is useful for the development of both transformations. For all three PSM-MT engines, the PIM-MT to PSM-MT transformation is executed in two steps: the first step is the transformation of the PIM-MT to an intermediate representation by ATL rules and the second step is the transformation of the intermediate representation to the textual format of the platform specific model transformation by an ATL query (see Figure 64).

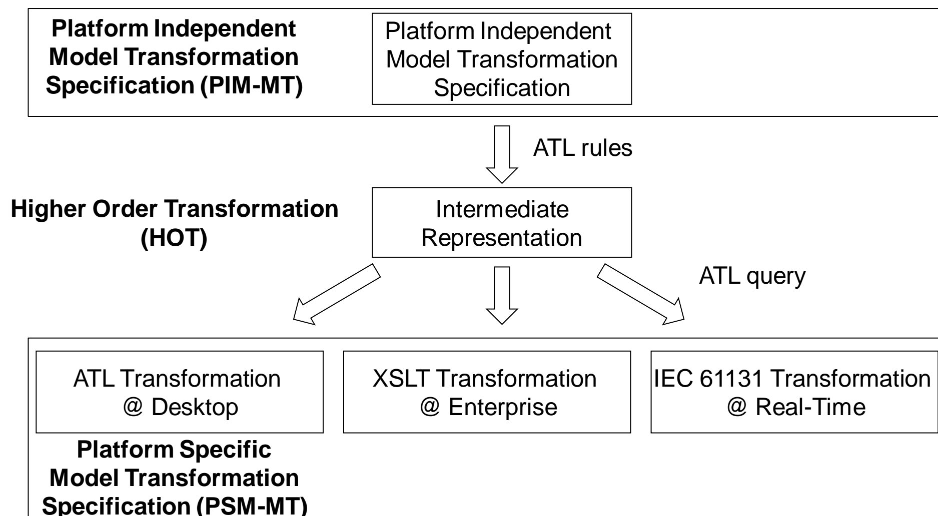

Figure 64: Implementation of the Higher Order Transformation (HOT)

The intermediate representation of the platform independent model transformation specification represents the elements of platform independent model transformation language. As specified in Section 5, this is a subset of the ATL language, which is used as a platform independent model transformation language. This intermediate representation eases the transformation to the textual representation of the platform specific model transformation languages by an ATL query.

The metamodel of the intermediate representation for platform independent model transformation specifications (PIM-MT) is shown in Figure 65. This metamodel defines the abstract syntax specification of the PIM-MT model transformation specification. A model transformation specification consists of a set of "Rule" elements included in a "Module", which holds all "Rule" elements belonging to a model transformation specification. A rule is identified by its "name" attribute. The execution of a "Rule" maps a single "InPattern" element to multiple "OutPattern" elements. Both elements, "InPattern" and "OutPattern", include a "type" attribute, which specifies the type of model elements referenced by the pattern. The type information is a characteristic of model transformation languages, which assume metamodels with type classification. Therefore, the type information is part of the intermediate model transformation specification and must be mapped to the platform specific metamodels.

Within the "InPattern", the type is the first part of the pattern definition for matching elements of the input model. The second part of the pattern definition is an optional filter, which is specified by a "HelperCall" to a user defined function. A user defined function used as a filter for an "InPattern" must return a Boolean value indicating a positive or negative match for the element of the input model. A "HelperCall" can include multiple arguments. As specified in Section 5, user defined functions referenced by "HelperCall" are used to avoid the transformation of complex expressions by the PIM-MT to PSM-MT transformation. Therefore, only "SimpleExpression" elements are allowed as arguments for a helper call. Complex expressions are hidden within the platform specific implementation of the user defined functions. A "SimpleExpression" is either a reference to an attribute of the input model element matched by the "InPattern" or a constant string value, specified by the "type" attribute of the "SimpleExpression". The "value" attribute holds either the name of the referenced attribute or the constant string value.

Each "OutPattern" element creates a target model element selected by the "type" attribute of the "OutPattern" element. The attributes of created target elements can be optionally specified by the "Binding" elements of the "OutPattern" element. Each "Binding" specifies a value for an attribute of the target model element by a "SimpleExpression" or a "HelperCall", which were already introduced for the "InPattern" element.

The user defined functions referenced by "HelperCall" elements are not part of the intermediate representation of the platform independent model transformation specification. Although being implemented as user defined functions within the platform specific model transformation engine, they are in-situ available to the platform independent model transformation specification similar to system functions. Function declarations of user defined functions referenced by "HelperCall" elements could enable static checks of the function signature by the PIM-MT to PSM-MT higher order transformation. Unfortunately, such a static check is too complex for the model transformation technology currently available and therefore omitted from the intermediate representation of the platform independent model specification. Nevertheless, the signature of user defined functions can be checked by the editor used

for the platform independent model transformations (for example the ATL editor in the evaluation of this thesis).

The implementation of the transformation from the platform independent model transformation specification to the intermediate model transformation specification is listed in the appendix, Section 9.1.

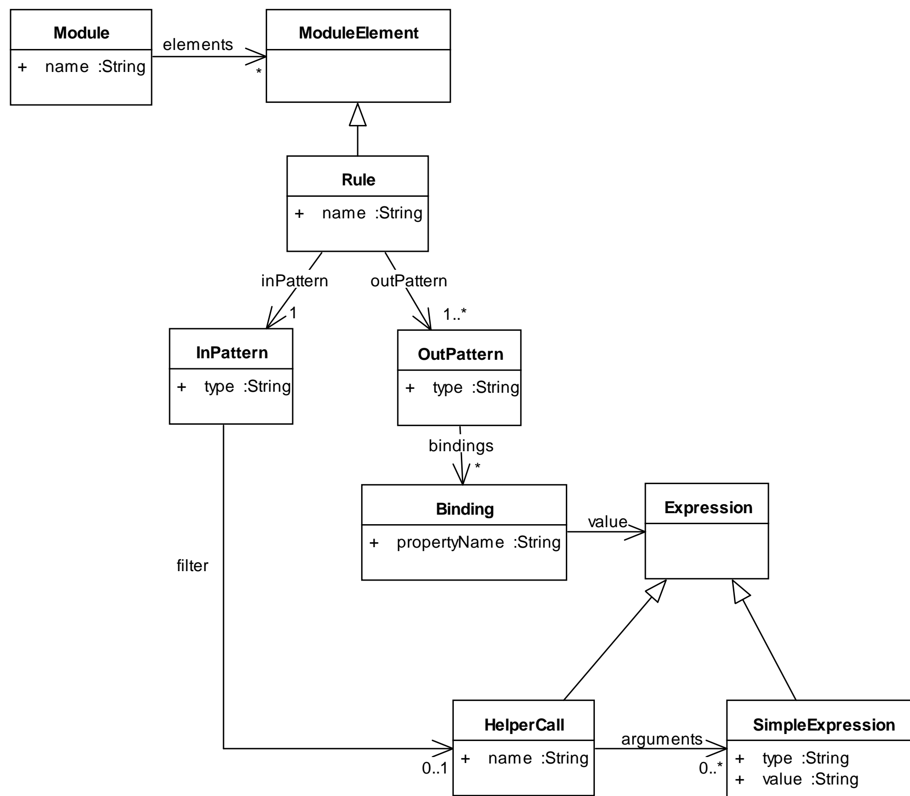

Figure 65: PIM-MT Intermediate Representation

While the PIM-MT to PSM-MT transformation uses the same technology for all platform specific model transformation specifications (ATL rules and ATL query), the evaluation of the metamodel transformation in the opposite direction uses different technologies for the platform specific metamodels. These different technologies are required, since the platform specific metamodel representations in general are only available as textual representations, which must be transformed to the modeling technologies used by platform independent model transformation specifications by a text to model transformation.

For desktop model transformations, no metamodel transformation is required, since ATL is used for both the platform independent model transformation specification and as a desktop model transformation engine. The transformation of metamodel definitions based on XML schema (XSD) used for enterprise model transformations is already provided by the Eclipse Modeling Framework (EMF) toolkit[28]. For the transformation of the metamodel used by the runtime model transformation engine implemented as part of this thesis, a parser for the structured text programming language of IEC 61131-3 [In03b] was implemented as part of the evaluation to transform the metamodel definition based on IEC 61131-3 function blocks to an Ecore representation. This parser was implemented with the help of the Eclipse XTEXT framework[29], which provides support both for parser development and for metamodel definitions.

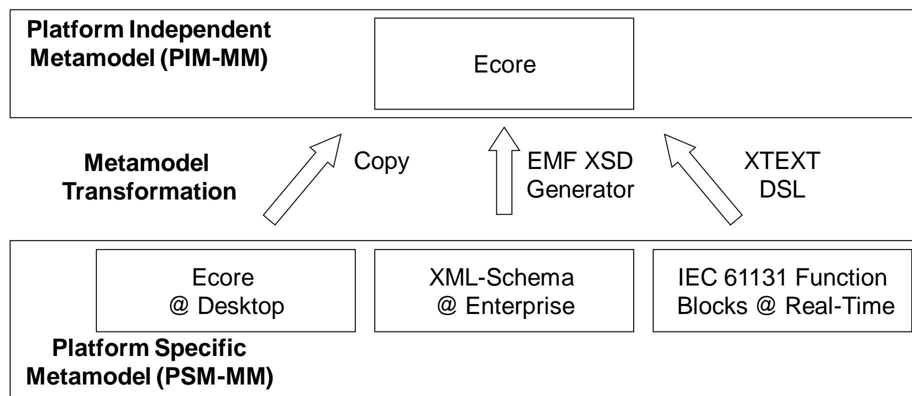

Figure 66: Implementation of the Metamodel Transformations

In the next sections, the platform independent model transformation specification used for evaluation is introduced followed by the implementations of PIM-MT to PSM-MT transformations for desktop, enterprise, and runtime model transformation specifications.

6.1 PIM-MT transformation specification example

For the evaluation of the PIM-MT to PSM-MT transformation, the model transformation specification for the reconfiguration of a labeling machine as introduced in Section 2 is used. Depending on the format of bottles, different labeling devices are connected to the labeling machine, for example a front labeling device and a back labeling device. The change of a configuration is detected by the electrical devices connected to the machine controller and requires a reconfiguration of the software, which controls the electrical devices of the labeling device. Therefore, a model transformation from the electrical model to the automation model of the currently active machine configuration is required.

---

[28] http://www.eclipse.org/modeling/emf/
[29] http://www.eclipse.org/Xtext/

The platform independent electrical metamodel used for evaluation is shown in Figure 67. The electrical devices controlling a labeling unit are the “DriveControlUnit”, which is a frequency converter. The “DriveControlUnit” controls one or more axis represented by a “DriveObject”, which are referenced by the “driveObject” attribute. The “DriveObject” is identified within the I/O communication by its “address” attribute. Special “DriveControlUnit” products available by an automation provider are the “S120DriveControlUnit” and the “G120DriveControlUnit”. The “S120ControlUnit” can specify its power circuit by the “inFeed” attribute.

All electrical elements share a “name” attribute and a status attribute named “active” by their common super-class “ElClass”. The “ElClass” and the “ElModel”, which aggregates all model elements, are not required from a technological point of the view. They are introduced in the platform independent electrical metamodel to ease the handling of the models within the Eclipse modeling environment.

The electrical metamodel used in the evaluation demonstrates the key features of machine engineering models as presented in Section 3.5: attributes, super-types, and references represented by attributes. Therefore, the example used in the evaluation can be generalized to production machines with more elaborated and bigger metamodels.

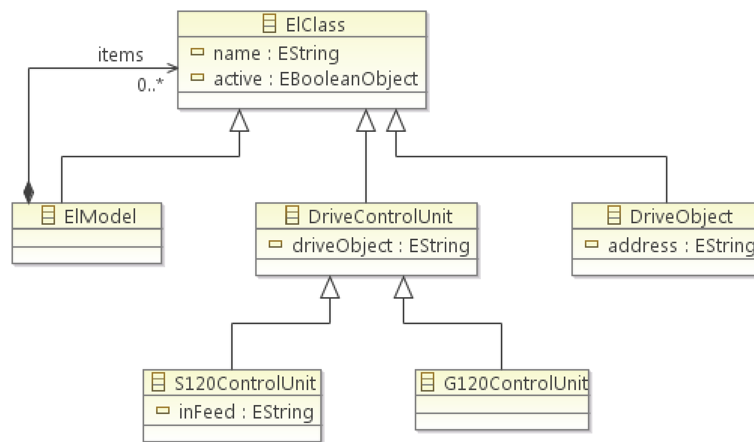

Figure 67: Platform Independent Electrical Metamodel (Ecore)

The automation metamodel used as a target metamodel in the platform independent model transformation is shown in Figure 68. It includes the programming elements required by an IEC 61131 [In03b] programmable logic controller (PLC) to control the labeling devices: the function blocks “LabelDeviceCuControl” and “LabelDeviceControl”, and the PLC operating system elements “TechnologyObject” and “DriveAxis”. The “TechnologyObject” is provided by the PLC operating system to support motion specific operations of production machines. For example, the “DriveAxis” as a specialized technology object, provides functions to program speed controlled axes. From the automation metamodel point of view, all elements (application program specific or PLC operating system specific) are handled in the same way by the

model transformation specification. It is up to a platform specific implementation to map these model elements to the elements of a specific implementation environment. A "LabelDeviceCuControl" references "LabelDeviceControl" elements by its "labelDevice" attribute. The "TechnologyObject" accesses the associated electrical device by its "address" attribute.

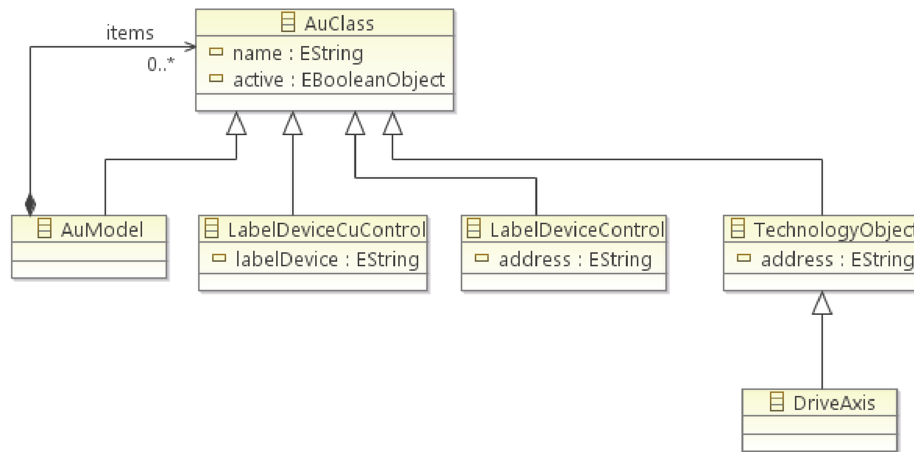

Figure 68: Platform Independent Automation Metamodel (Ecore)

In the following sections, the example platform independent transformation specification from the electrical model to the automation model shown in Figure 69 is transformed into the three platform specific model transformation specifications: desktop, enterprise, and real-time model transformations.

The example consists of two transformation rules. The first rule "do2to" demonstrates the usage of filters for the source pattern, the usage of helper functions "RDLabelDeviceControl" and "RDTechnologyObject" for the reference designators of the target pattern elements, and the assignment of source attribute values to target attributes for "address". The second rule "cu2control" shows the resolution of references by the helper function "RDLabelDeviceControl" instead of ATL traceability links for the attribute "labelDevice".

```
-- @path MMELECTRICAL=/pim_mt/model/electrical.ecore
-- @path MMAUTOMATION=/pim_mt/model/automation.ecore

module electrical2automationStringRef;
create OUTAUTOMATION : MMAUTOMATION from INELECTRICAL :
MMELECTRICAL;

rule do2to
{
from
	s: MMELECTRICAL!DriveObject
	(
		thisModule.MatchString(s.name, 'frontLabel')
	)
to
	u: MMAUTOMATION!LabelDeviceControl
	(
		name <- thisModule.RDLabelDeviceControl(s.name)
	)
	, technologyObject: MMAUTOMATION!TechnologyObject
	(
		name <- thisModule.RDTechnologyObject(s.name)
		, address <- s.address
	)
}

rule cu2control
{
from
	s: MMELECTRICAL!DriveControlUnit
to
	t: MMAUTOMATION!LabelDeviceCuControl
	(
		name <-
thisModule.RDLabelDeviceCuControl(s.name)
		, labelDevice <-
thisModule.RDLabelDeviceControl(s.driveObject)
	)
}
```

Figure 69: Platform Independent Model Transformation (PIM-MT) Example

6.2 Desktop Model Transformations - ATL to ATL

Model transformations executed on a local desktop are a common environment for the design and evaluation of engineering model transformation specifications. The ATL model transformation environment was chosen within this thesis for the platform independent specification of model transformations (PIM-MT), but can be also used as a platform specific model transformation engine (PSM-MT).

The PIM-MT to PSM-MT transformation from ATL to ATL can also be used to verify if the PIM-MT model transformation specification does not include modeling concepts, which cannot be handled by the PIM-MT to PSM-MT higher order transformation. This approach avoids the development of a special PIM-MT development environment with static checks of the PIM-MT model transformation specification, but still allows for the verification of the PIM-MT model transformation.

The evaluation setup for desktop model transformations is shown in Figure 70. In general, the source engineering model and the target engineering model are exchanged with the engineering tools by a tool adapter. The effort for the development of a tool adapter depends on the interfaces available for an engineering tool and from the complexity of the mapping of the engineering tools metamodel and engineering model to the models required by the desktop model transformation engine. For the application example of a bottle labeling machine, a tool adapter for an ECAD engineering tool and a tool adapter for a PLC engineering tool is required. In contrast to enterprise model transformations, where these adapters are provided as part of a PLM system, the adapters required for desktop model transformations must be developed especially for the desktop model transformation engine.

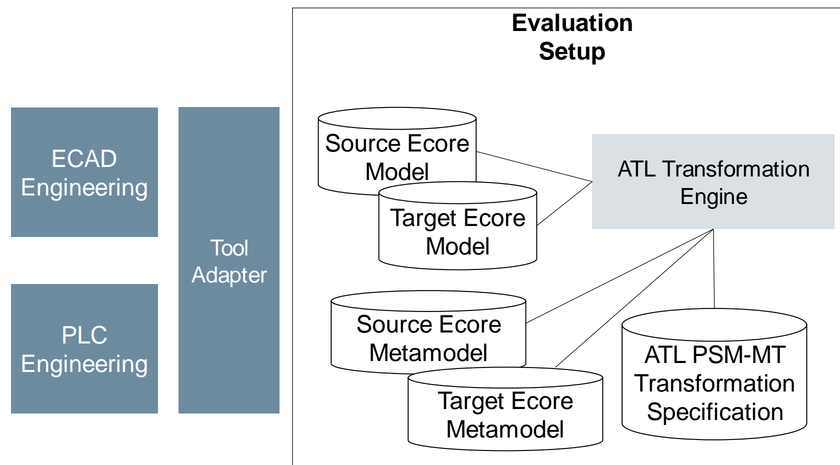

Figure 70: Desktop Model Transformation Evaluation

To avoid the development of a tool adapter, within the evaluation of this thesis, the source engineering metamodels and the target engineering metamodels together with the source models used for the test of the desktop model transformations have been created manually as Ecore models. The evaluation setup uses the Eclipse ATL model transformation engine[31] for the execution of the desktop model transformation specification.

6.2.1 PSM-MM to PIM-MM Transformation

Within the evaluation setup for desktop model transformations, the platform specific metamodels (PSM-MM) the electrical engineering model and the automation model were the same metamodels as the platform independent metamodels presented in Section 6.1. Therefore, the PSM-MM to PIM-MM transformation is a simple copy of the metamodel. In general, even if a tool adapter already provides an Ecore metamodel, a PSM-MM to PIM-MM transformation is required to generalize and simplify the tool dependent platform specific metamodel.

[31] http://www.eclipse.org/atl/

6.2.2 Model Instances

The model instances used for the evaluation of the desktop model transformation are specified in the XMI format [IS05], which is used by the Eclipse modeling platform for both the storage of metamodels and for the storage of metamodel instances.

For the source engineering model, the instance of the electrical metamodel shown in Figure 71 defines the electrical configuration of a labeling machine with a front labeling and a back labeling device. Both devices use a “DriveObject” as an inverter for their motor. The control unit assigned to the “DriveObject” is a general “DriveControlUnit” for the back labeling device and a special “S120ControlUnit” for the front labeling device.

```
<?xml version="1.0" encoding="ASCII"?>
<org.mtmda.electrical:ElModel
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:org.mtmda.electrical="http://electrical.mtmda.org"
    xsi:schemaLocation="http://electrical.mtmda.org
electrical.ecore"
    name="labelingMachine">
  <items xsi:type="org.mtmda.electrical:DriveObject"
      name="backLabel"
      address="0.1"/>
  <items xsi:type="org.mtmda.electrical:DriveObject"
      name="frontLabel"
      address="0.2"/>
  <items xsi:type="org.mtmda.electrical:DriveControlUnit"
      name="backLabelCU"
      driveObject="backLabel"/>
  <items xsi:type="org.mtmda.electrical:S120ControlUnit"
      name="frontLabelCU"
      driveObject="frontLabel"
      inFeed="1000W"/>
</org.mtmda.electrical:ElModel>
```

Figure 71: Electrical Model Example

The automation model example shown in Figure 72 was created as by a desktop model transformation from the electrical model example shown in Figure 71. It includes function blocks and technology objects, which control the electrical devices specified by the electrical engineering model.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<org.mtmda.automation:AuModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:org.mtmda.automation="http://automation.mtmda.org">
  <items xsi:type="org.mtmda.automation:LabelDeviceControl"
         name="fbrd_backLabel"/>
  <items xsi:type="org.mtmda.automation:LabelDeviceControl"
       name="fbrd_frontLabel"/>
  <items xsi:type="org.mtmda.automation:LabelDeviceCuControl"
       name="fbrd_backLabelCU"
       labelDevice="fbrd_backLabel"/>
  <items xsi:type="org.mtmda.automation:LabelDeviceCuControl"
       name="fbrd_frontLabelCU"
       labelDevice="fbrd_frontLabel"/>
  <items xsi:type="org.mtmda.automation:DriveAxis"
       name="tord_backLabel"/>
  <items xsi:type="org.mtmda.automation:TechnologyObject"
       name="tord_frontLabel"/>
</org.mtmda.automation:AuModel>
```

Figure 72: Automation Model Example

6.2.3 PIM-MT to PSM-MT Transformation

The higher order transformation from the platform independent model transformation specification (PIM-MT) to the ATL platform specific model transformation specification (PSM-MT) is implemented as an ATL query from the intermediate model transformation specification presented at the start of this section (see Figure 64).

Using the ATL both as a language for the platform independent specification of model transformations and as an engine for the platform specific execution of model transformation means that the PIM-MT to PSM-MT higher order transformation mainly discards concepts that are not part of the platform independent model transformation specification defined in Section 5 but are allowed within the ATL language. Therefore, the generated PSM-MT can be compared to the PIM-MT to reveal forbidden ATL language features.

The implementation of the PIM-MT to desktop PSM-MT higher order transformation is listed in Section 9.2

6.2.4 Rule Execution

The ATL desktop transformation engine does not support the specification of the execution order of the declarative model transformation rules used within this thesis. For typical desktop model transformation application scenarios, the explicit specification of the rule execution order is not required, since the model rule execution is not influenced by the system outside the model transformation as for enterprise model transformations or real-time model transformations.

6.2.5 Summary

The PIM-MT to PSM-MT environment for desktop model transformations is summarized in Figure 73. It is especially simple, because ATL is used for both platform independent and platform specific specifications of model transformations. The transformation of the platform specific metamodels to the platform independent metamodels is a simple copy. The PIM-MT specification is transformed by an ATL higher order transformations to the platform specific representation. This representation is executed as a desktop model transformation by the ATL desktop model transformation engine.

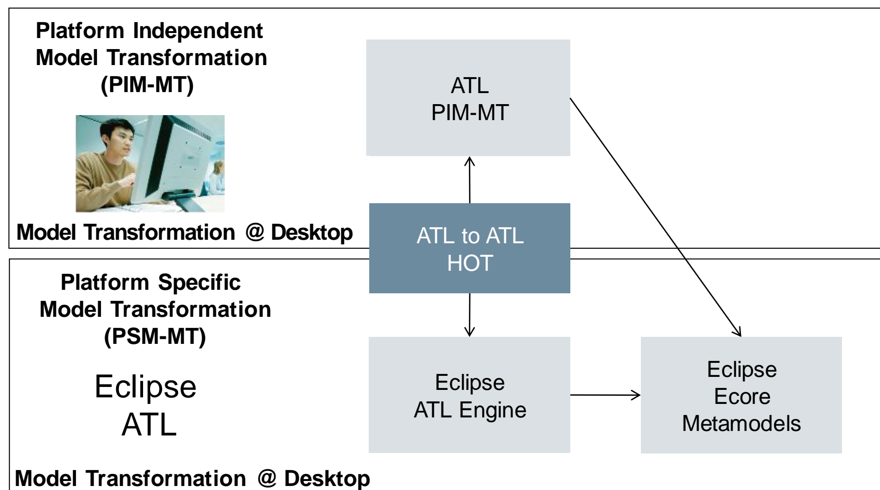

Figure 73: Desktop PIM-MT to PSM-MT Higher Order Transformation (HOT)

6.3 Enterprise Model Transformations - ATL to XSLT

For the execution of enterprise model transformations the Siemens Teamcenter[32] PLM system has been used as an example environment. A Teamcenter installation within an enterprise consists of a 4-tier installation as shown in Figure 74. The client tier includes the authoring applications to build the engineering models used in machine engineering. For the application example of the bottle labeling machine, this is an ECAD engineering application like EPLAN Electric[33] for the electrical engineering model and Siemens SIMOTION SCOUT [Si08a] for automation engineering. The client tier is connected to the web tier of the Teamcenter installation, which routes client requests to the business logic of the enterprise tier. The web tier is typically implemented by an application server like JBoss[34], which routes client requests to the business logic and provides services message handling and message transformations. The enterprise tier hosts the PLM business logic and defines the data schemata. The resource tier provides persistence for databases and files.

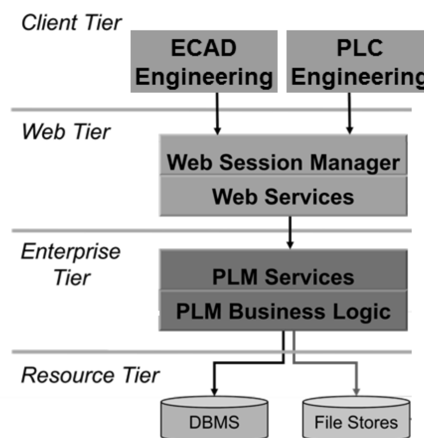

Figure 74: Siemens Teamcenter 4-tier Architecture[35]

The evaluation of enterprise model transformation uses the web tier and the enterprise tier of a Teamcenter installation. The enterprise tier provides the schemata used for the definition of engineering models within a PLM environment (see Figure 75). For the execution of an enterprise model transformation, a source model is created as an XML document based on the PLM schema of an electrical engineering model. This source model is processed by an XSLT transformation, which was created as a platform specific model transformation (PSM-MT) from the platform independent model transformation (PSM-MT) by a higher order ATL transformation as part of the work this thesis. The

[32] http://www.plm.automation.siemens.com/en_us/products/teamcenter/
[33] http://www.eplan.de/en/solutions/electrical-engineering/eplan-electric-p8/
[34] https://www.jboss.org
[35] http://www.plmworld.org

platform specific model transformation must be XML schema aware. Therefore, the Altova XML engine[36], which is one of the few available schema aware XSLT engines, was used for the platform specific model transformation. The target model created by the platform specific model transformation is based on the PLM schema of an automation engineering model. Both PLM schemata, electrical and automation engineering model, were defined for the evaluation as an extension of PLMXML schema provided by Siemens Teamcenter[37].

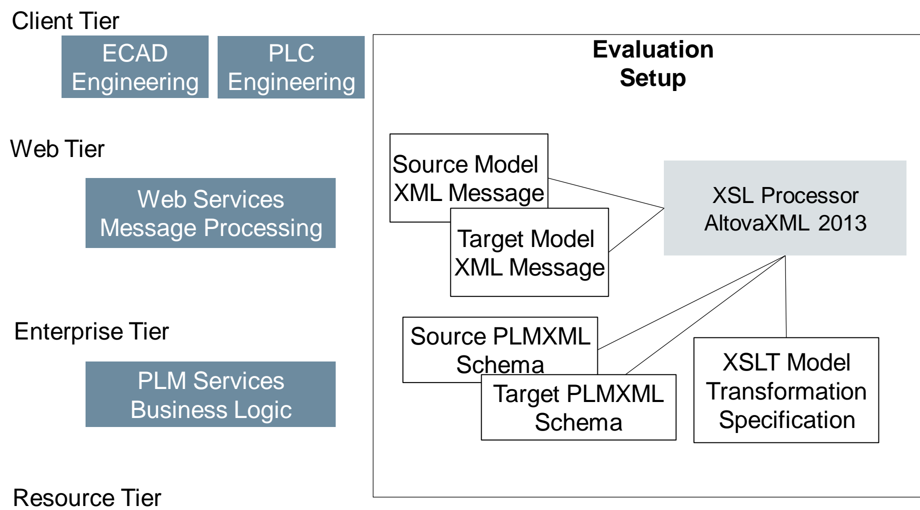

Figure 75: Enterprise Model Transformation Evaluation

6.3.1 PSM-MM to PIM-MM Transformation

For the electrical engineering metamodel and the automation engineering metamodel used in the application example of a bottle labeling machine, two extensions of the Siemens Teamcenter PLMXML schema have been defined. The elements of an engineering model are handled in Teamcenter as so called “Items”, which can be used in different structures, e.g. in a product structure or in a bill of material (BOM). As presented in Section 4.2.2, these elements are extensions of the “StructureBase” type of the PLMXML schema. For example, the “DriveControlUnit” element and the “S120ControlUnit” element of the electrical engineering schema shown in Figure 76 extend the “StructureBase” type by using the “xsd:extension” attribute.

[36] http://www.altova.com/altovaxml.html
[37] http://www.plm.automation.siemens.com/en_us/products/open/plmxml/schemas.shtml

```
  <!-- -->
  <!-- ****** DriveControlUnit ********************** -->
  <!-- -->
  <xsd:complexType name="DriveControlUnit">
    <xsd:annotation>
      <xsd:documentation>
      Represents a hardware device.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="plmxml:StructureBase">
       <xsd:attribute name="driveObject" type="xsd:string"
use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="DriveControlUnit"
type="plmxml:DriveControlUnit"
substitutionGroup="plmxml:Structure"/>
  <!-- -->
  <!-- ****** S120ControlUnit *********************** -->
  <!-- -->
  <xsd:complexType name="S120ControlUnit">
    <xsd:annotation>
      <xsd:documentation>
      Represents a hardware device.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="plmxml:DriveControlUnit">
       <xsd:attribute name="inFeed" type="xsd:string"
use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="S120ControlUnit"
type="plmxml:S120ControlUnit"
substitutionGroup="plmxml:Structure"/>
```

Figure 76: XML Schema Platform Specific Metamodel –
Electrical Engineering Metamodel

For the transformation of the platform specific engineering metamodels based on PLMXML schema (PSM-MM) to the platform independent Ecore models (PIM-MM), the XSD to Ecore generator provided as part of the Eclipse EMF framework can be used [St09]. As an example, Figure 77 shows the Ecore PIM-MM generated from the

electrical engineering PSM-MM. In addition to the elements of the electrical engineering metamodel, the generated PIM-MM includes additional elements from the platform specific metamodel, e.g. the “StructureBase” element and all of its generalization elements. In comparison to the general PIM-MM presented as an example in Figure 67, all elements of the PIM-MM, which are used as part of the model transformation rules, must be compatible for interchangeable use by the platform independent model transformation specification. For example, the “name” attribute is available as part of the “DescriptionBase” element for the enterprise electrical metamodel as well as in the “ElClass” element for the electrical engineering metamodel shown in Figure 67.

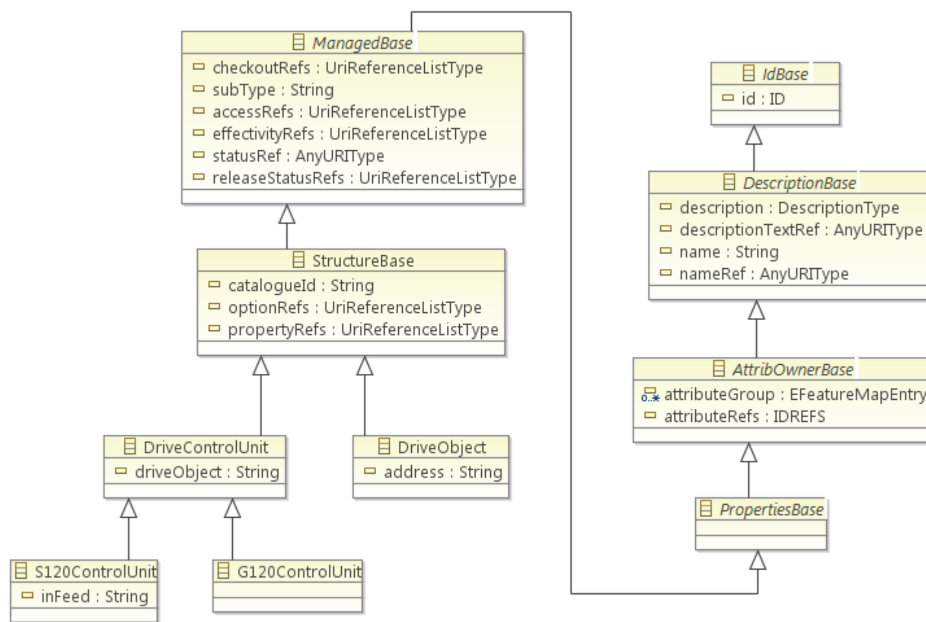

Figure 77: Ecore Platform Specific Metamodel – Electrical Engineering Metamodel

6.3.2 Model Instances

Within enterprise model transformations, in general not a complete engineering model is handled by model transformations but only parts of the engineering models shall be reconciled by engineering model transformation. The handling of a complete engineering model might be impossible due to the sheer size of the engineering models. But also organizational restrictions require the transformation of parts of an engineering model. Parts of the engineering model might be locked due to release responsibilities or due to access rights. The use of weak references as defined in Section 5.3 allows for the execution of model transformations even if elements of other parts of an engineering model are not accessible or even not available yet.

Therefore, a model instance handled as a message within the web tier of a Teamcenter installation conforms to its PLMXML schema but reflects only a sub-part of the complete engineering model. An example instance of an electrical engineering model message is shown in Figure 78. It includes the drive devices and the drive controllers of the front labeling device and the back labeling device of the bottle labeling machine.

```
<?xml version="1.0" encoding="UTF-8"?>
<plmxml:PLMXML author="" date="2001-01-01"
schemaVersion="0.0" time="12:00:00"
xmlns:plmxml="http://www.plmxml.org/Schemas/PLMXMLSchema"
xmlns:xhtml="http://www.w3.org/1999/xhtml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.plmxml.org/Schemas/PLMXMLSchem
a PLMXMLElectricalSchema.xsd "
>
      <plmxml:DriveObject address="1.0" name="frontLabel" />
      <plmxml:DriveObject address="2.0" name="backLabel" />
      <plmxml:DriveControlUnit driveObject="frontLabel"
            name="frontLabelCU" />
      <plmxml:S120ControlUnit driveObject="backLabel"
            inFeed="1KW" name="backLabelCU" />
</plmxml:PLMXML>
```

Figure 78: XML Platform Specific Model Instance

6.3.3 PIM-MT to PSM-MT Transformation

The higher order transformation from the platform independent model transformation specification PIM-MT to the XSLT platform specific model transformation specification is implemented as an ATL query from the intermediate model transformation specification presented at the start of this section (see Figure 64). The ATL query creates an XSL style sheet with XSL transformations for each rule of the platform independent model transformation specification (PIM-MT). For the handling of specialization relationships, the schema aware XSLT engine must access the schema definition of the source model. Therefore, the XSLT specification of the PSM-MT transformation requires access to the schema definition of the source engineering model by an "xsl:import-schema" declaration (see top of Figure 79). The schema definition of the target engineering model is imported to check the structure of the generated target model elements. Secondly, the XSLT specification imports the user-defined functions, which are not generated by the higher order transformation but are manually programmed for the platform specific execution engine by the "xsl:include" declaration.

Each platform independent transformation rule is transformed in an XSLT template rule. Matching of the source model elements is defined by the match attribute of the template rule, which is a pattern of the nodes to which the rule applies. The "InPattern" of the platform independent model transformation is transformed into two elements of the match attribute. First, the element type is matched by an element test, which selects the type of the element defined within the source model schema. This element test is only available for schema aware XSLT processors. Second, an optional user defined function can check additional attributes of the matched source model element.

The target model elements are constructed by the sequence constructor of the XSLT template rule according to the "OutPattern" elements of the platform independent model transformation. As shown in Figure 79, the sequence constructor builds a set of nodes according to the types defined in the target model schema by the "xsl:element" instructions. The attributes of the created nodes are initialized by the "xsl:attribute" instruction together with user defined functions.

The implementation of the PIM-MT to enterprise PSM-MT higher order transformation is listed in Section 9.3.

```
<xsl:import-schema
namespace="http://www.plmxml.org/Schemas/PLMXMLSchema"
schema-location="../model/PLMXMLElectricalSchema.xsd " />
<xsl:import-schema
namespace="http://www.plmxml.org/Schemas/PLMXMLSchema"
schema-location="../model/PLMXMLAutomationSchema.xsd " />
<xsl:include href="userdefinedfunctions.xsl"/>

<!-- rule do2to-->
<xsl:template match="element(*, plmxml:DriveObject)
[plmxml:MatchString(@name, 'frontLabel')]">
<!-- OutPattern -->
<xsl:element name="plmxml:LabelDeviceControl"
type="plmxml:LabelDeviceControl">
      <xsl:attribute name="name"><xsl:value-of
            select="plmxml:RDLabelDeviceControl(@name)"
/></xsl:attribute>
</xsl:element>
<!-- OutPattern -->
<xsl:element name="plmxml:TechnologyObject"
type="plmxml:TechnologyObject">
      <xsl:attribute name="name"><xsl:value-of
            select="plmxml:RDTechnologyObject(@name)"/>
      </xsl:attribute>
      <xsl:attribute name="address"><xsl:value-of
            select="@address" />
      </xsl:attribute>
</xsl:element>
</xsl:template>
<!-- rule cu2control-->
<xsl:template match="element(*, plmxml:DriveControlUnit)">
<!-- OutPattern -->
      <xsl:element name="plmxml:LabelDeviceCuControl"
type="plmxml:LabelDeviceCuControl">
      <xsl:attribute name="name"><xsl:value-of
            select="plmxml:RDLabelDeviceCuControl(@name)" />
      </xsl:attribute>
      <xsl:attribute name="labelDevice"><xsl:value-of
      select="plmxml:RDLabelDeviceControl(@driveObject)"/>
      </xsl:attribute>
</xsl:element>
</xsl:template>
```

Figure 79: XSLT Specific Model Transformation Rule

6.3.4 Rule Execution

XSL transformations support the definition of the rule execution order by the “xsl:apply-templates” instruction. Starting from the initial template shown in Figure 80, all generated platform specific model transformation rules are executed. If required within a platform specific model transformation, the optional “select” attribute of the “xsl:apply-templates” instruction can select subsets of the source model elements for template rule application to define the execution order of model transformation rules. For the evaluation scenario used in this thesis, no specific rule execution order was required. In general, XSLT “xsl:apply-templates” instructions are execute in the order given by the sequence constructor of the initial template in an XSLT specification.

```
<xsl:template match="/*">

<plmxml:PLMXML
       author=""
       date="2001-01-01"
       schemaVersion="0.0"
       time="12:00:00"
       xmlns:plmxml=
       "http://www.plmxml.org/Schemas/PLMXMLSchema"
       xmlns:xhtml="http://www.w3.org/1999/xhtml"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
       "http://www.plmxml.org/Schemas/PLMXMLSchema
       ../model/PLMXMLAutomationSchema.xsd "
              >

       <xsl:apply-templates/>

</plmxml:PLMXML>
</xsl:template>
```

Figure 80: XSLT Rule Execution – Initial Template

6.3.5 Summary

The complete environment used for the transformation of a platform independent model transformation specification to an enterprise platform specific model transformation specification is summarized in Figure 81. The platform specific engineering metamodels used in the Teamcenter PLM environment on an enterprise server are defined by Teamcenter PLMXML schema definitions. These metamodels are transformed from platform specific metamodel representation (PSM-MM) to the platform independent metamodel representation (PIM-MM) by the XSD2Java plugin, which is provided as part of the Eclipse Modeling Framework EMF[38]. The PIM-MM Ecore representation generated by the XSD2Java plugin can be directly used by the platform independent model transformation specification (PIM-MT) without further adaptation, since the XML schema definition of the PSM-MM supports the same set features which is required by the PIM-MM Ecore representation.

The platform independent metamodel is used by the platform independent model transformation specification, which defines the transformation rules required for the reconfiguration of a bottle labeling machine. This platform independent rule specification is transformed by an ATL higher order transformation to an enterprise platform specific rule execution engine, which is executed by the Altova XML engine.

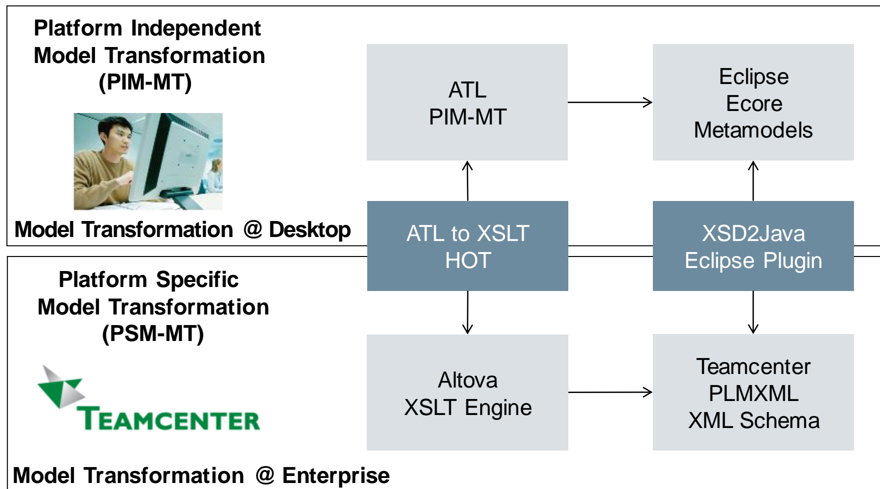

Figure 81: Enterprise PIM-MT to PSM-MT Higher Order Transformation (HOT)

[38] http://www.eclipse.org/modeling/emf/

6.4 Real-Time Model Transformations - ATL to IEC 61131-3

For the execution of model transformations on standard real-time IEC 61131-3 [In03b] programmable logic controllers (PLC), an IEC 61131-3 platform specific metamodel (PSM-MM) and an IEC 61131-3 platform specific model transformation engine was developed as part of this thesis. Both are based on the structured text (ST) programming language [In03b]. The ST language is a structured programming language with program organization units (POU) called programs, function blocks, and functions as programming elements.

The structure of the real-time model transformation engine implementation is shown in Figure 82. Without the real-time model transformation engine, an automation program consists of the IEC 61131-3 automation system provided by the PLC vendor together with the machine application library and user program developed by the machine builder (the lower building blocks in Figure 82). The real-time model transformation adds two additional building blocks on top of the machine automation program: the metamodel and the model instances of the engineering models implemented as function blocks together with the rule function blocks generated from the platform independent model transformation specification by the higher order PIM-MT to PSM-MT transformation.

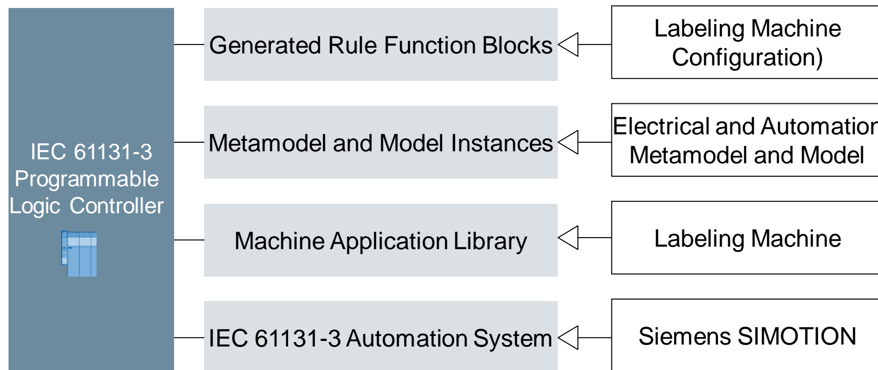

Figure 82: Platform Specific Model Transformation Language – IEC 61131-3 Real-Time Systems

The evaluation environment used for real-time model transformations is shown in Figure 83. The test of the model transformation engine on a bottle labeling machine involves the machine mechanics, the electrical I/O devices and drives interacting with the machine mechanics, and the PLC controlling these electrical I/O devices and drives. A desktop PC is used as an engineering station for programming and monitoring the PLC. For the work of this thesis, no real bottle labeling machine was available. Therefore, the execution of the real-time model transformations was tested running on

the hardware of a Siemens SIMOTION D435 programmable logic together with an engineering station omitted (shown at the right side of Figure 83). The electrical I/O devices and the machine mechanics were omitted (shown at the left side of Figure 83). Changes of the electrical I/O devices and drive reflecting a change of the machine configuration were simulated by manual modifications of the electrical engineering model within the automation controller. The execution of the platform specific model transformation rules on the SIMOTION PLC can be monitored within a log file on the engineering station. The log file is created by a message listener on the engineering station, which receives debug messages by a TCP/IP connection from the real-time rule execution engine.

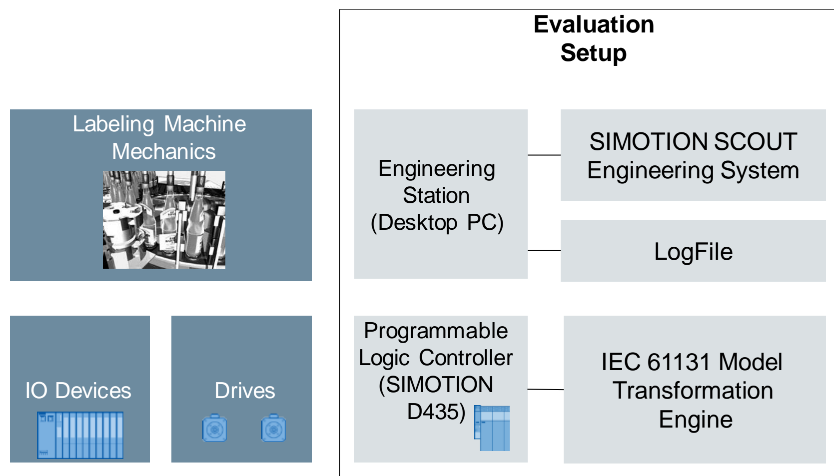

Figure 83: Real-Time Model Transformation Evaluation Setup

The real-time model transformation engine is integrated in the SIMOTION controller as a structured text (ST) code library. Figure 84 shows at the right side a screen shot of the SIMOTION SCOUT engineering system and at the left as an enlargement the elements of the model transformation library. The type definitions of the electrical metamodel and the automation metamodel are included in the “automationTypes” and “electricalTypes” ST units. Based on these type definitions, the units “automationInstances” and “electricalInstances” hold the model instances of the electrical engineering model and the automation engineering model. Finally, the “ModelRules” together with the “UserDefinedFunctions” include the implementation of the real-time model transformation rules. The model transformation rules are called by a program, which is executed by a motion task of the SIMOTION execution system. In the next sections, the elements of the real-time model transformation execution engine shown in Figure 84 are explained in detail.

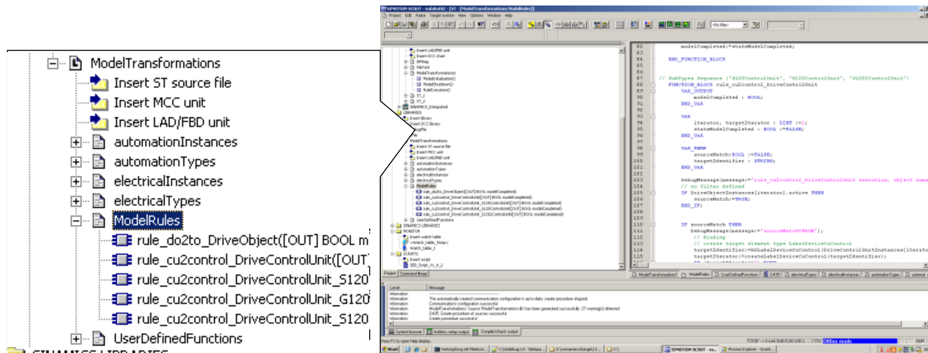

Figure 84: ATL Rule to IEC 61131-3 Structured Text - Model Transformations Library

6.4.1 PSM-MM to PIM-MM Transformation

The engineering metamodels are specified within the SIMOTION automation controller as IEC 61131-3 function blocks. These function blocks can be created manually as part of the machine application development, can be created by using scripting functions within the SIMOTION SCOUT development environment, or can be provided as part of a machine application library. The function blocks belonging to the same engineering metamodel are grouped in a common unit in the SIMOTION SCOUT engineering environment, e.g. the unit "electricalTypes" for the electrical metamodel and "automationTypes" for the automation metamodel.

As an example from the electrical engineering metamodel, the function blocks "DriveControlUnit" and "S120ControlUnit" are shown in Figure 85. Each function block from the platform specific engineering metamodel represents an Ecore class model element in the platform independent engineering metamodel with the same name as the function block. The input variables of the function blocks represent the Ecore attributes of the Ecore class, for example the input variables "name", "active", and "driveObject" are transformed to Ecore attributes with the same name and data type. Generalization relationships cannot be expressed in the current edition of IEC 61131-3. For example, the "S120ControlUnit" metamodel element shall be a specialization of the "DriveControlUnit" metamodel element. It inherits the attributes "name", "active", and "driveObject" attributes. In the IEC 61131 metamodel representation, these attributes are duplicated in the specialized class as shown in Figure 85. Therefore, generalization relationships must be manually added on the level of the platform independent metamodel if required. In the future, the upcoming third edition of IEC 61131-3 [In12] supports generalization relationships between function blocks, which can be used in PSM-MM to PIM-MM transformation.

```
FUNCTION_BLOCK DriveControlUnit
    VAR_INPUT
        name : STRING; // EString
        active : BOOL; // EBooleanObject
        driveObject : STRING; // EString
    END_VAR

    VAR_OUTPUT
        result: DINT;
    END_VAR;


    DebugMessage(message:='execute DriveControlUnit FB',
parameter:=name);
    result:=0;
    IF NOT active THEN
        GOTO block_exit;
    END_IF;

    result:=1;


    block_exit:
    ;
END_FUNCTION_BLOCK

FUNCTION_BLOCK S120ControlUnit
    VAR_INPUT
        name : STRING; // EString
        active : BOOL; // EBooleanObject
        driveObject : STRING; // EString
        inFeed : STRING; // EString
    END_VAR

    VAR_OUTPUT
        result: DINT;
    END_VAR;

...

END_FUNCTION_BLOCK
```

Figure 85: IEC 61131-3 Platform Specific Metamodel

The implementation of the PSM-MM to PIM-MM transformation for the IEC 61131-3 model transformation platform is shown in Figure 86. The function blocks of the engineering model can be exported from the SIMOTION SCOUT engineering system

available as a textual representation of the IEC 61131-3 structured text (ST) programming language. This textual representation is transformed by a text to model transformation to an Ecore model, which represents the abstract syntax tree of the structured text file. The text to model transformation was implemented with the help of the Eclipse XTEXT framework[39] as part of the work of this thesis and a master thesis [Ge12]. The abstract syntax tree of the structured text programming file is too detailed to be used as a platform independent metamodel (PIM-MM). Therefore, a metamodel transformation implemented with ATL extracts the function blocks and input variables and transforms them to the platform independent metamodel as described above (see Section 9.4). Finally, generalization relationships are added manually by a second ATL transformation listed in Section 9.5.

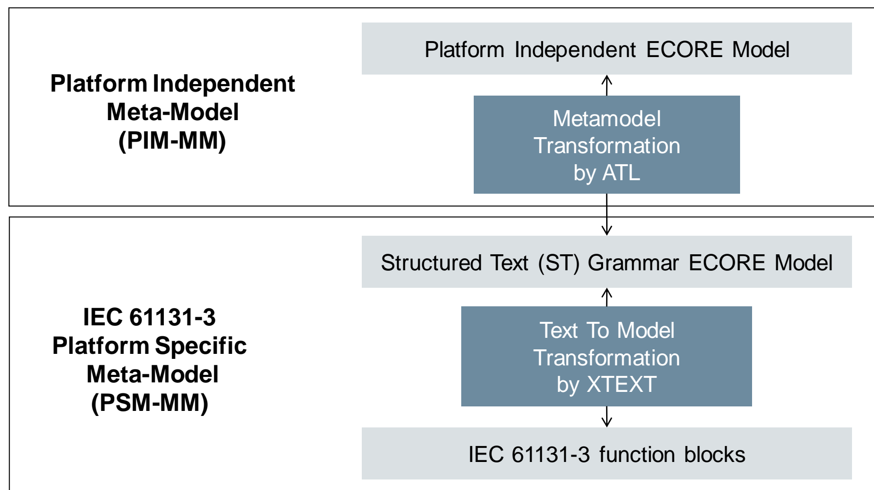

Figure 86: IEC 61131-3 PSM-MM to PIM-MM Transformation

[39] http://www.eclipse.org/Xtext/

6.4.2 Model Instances

The engineering model instances created from IEC 61131-3 metamodel elements described in the previous section are stored in additional library units, e.g. the unit "electricalInstances" for the electrical engineering model and "automationInstances" for the automation metamodel. Similar to the metamodel elements, the model instances can be created manually or can be generated automatically within the SIMOTION SCOUT engineering system.

Each engineering model element is an instance of a function block from the engineering metamodel. Due to restrictions of the IEC 61131-3 programming environment, which does not support generalization, the engineering model elements cannot be stored within a common data structure but must be split to multiple arrays with a separate array for each metamodel function block. For example, all instances of the metamodel element "DriveObject" are stored in the array "DriveObjectInstances", all instances of the metamodel element "DriveControlUnit" are stored in the array "DriveControlUnitInstances" and so on (see Figure 87). These data structures are used as part of the PIM-MT to PSM-MT higher order transformation described in the next section.

Beside the arrays holding the model instances of each metamodel element, a factory function to create model instances is required for each metamodel element, e.g. "createDriveObject" for the "DriveObject" metamodel type. The creation of model instances is handled specially within real-time controllers. The SIMOTION programmable logic controllers do not support the dynamic creation of elements. The reason for this is the required predictability of memory consumption and execution performance for real-time control programs. Therefore, the real-time model transformation engine implemented as part of this thesis does not create model elements dynamically but activates elements from a given set of predefined elements. This strategy is common for implementation of modular machinery as the bottle labeling machine used as an application example to guarantee the real-time execution constraints for all possible machine configurations. The create factory methods implement this activation strategy, if the model transformation rules request the creation of a model instance element.

```
UNIT electricalInstances;
INTERFACE
    USES electricalTypes;

    FUNCTION InitelectricalObjects;
    FUNCTION DebugelectricalObjects;

    FUNCTION createDriveObject;
    FUNCTION createDriveControlUnit;
    FUNCTION createS120ControlUnit;
    FUNCTION createG120ControlUnit;
    FUNCTION createS1202ControlUnit;

    VAR_GLOBAL
        DriveObjectInstances : ARRAY[1.. DriveObjectNum]
OF DriveObject;
        DriveControlUnitInstances : ARRAY[1..
DriveControlUnitNum] OF DriveControlUnit;
        S120ControlUnitInstances : ARRAY[1..
S120ControlUnitNum] OF S120ControlUnit;
        G120ControlUnitInstances : ARRAY[1..
G120ControlUnitNum] OF G120ControlUnit;
        S1202ControlUnitInstances : ARRAY[1..
S1202ControlUnitNum] OF S1202ControlUnit;
    END_VAR
END_INTERFACE
```

Figure 87: IEC 61131-3 Platform Specific Model Instances

6.4.3 PIM-MT to PSM-MT Transformation

The higher order transformation from the platform independent model transformation specification PIM-MT to the IEC 61131-3 platform specific model transformation specification is implemented as an ATL query from the intermediate model transformation specification presented at the start of this section (see Figure 64). The implementation of the real-time model transformation includes a unit "ModelRules", which holds the platform specific model transformation specification generated by the PIM-MT to PSM-MT higher order transformation, and a unit "UserDefinedFunctions", which holds the manually written functions used in source pattern matching or target pattern bindings.

For each rule of the intermediate representation of the platform independent model transformation, the ATL query creates rule execution function blocks. The first part of a rule execution function block checks the matching conditions for source model elements of the model transformation rule. The second part of a rule function block creates the target model elements together with their bindings. The example shown in Figure 88 is created by the PIM-MT to PSM-MT transformation from the rule example in Figure 69.

Matching of the source element type is handled by the generation of a rule execution function blocks for the source model elements type and for all subtypes of the source model element type. In the example, the source model element "DriveObject" has no subtypes. Therefore, only a single function block "rule_do2to_DriveObject" was created for this transformation rule. The source model element "DriveControlUnit" used as a source model element in the second transformation rule example has "S120ControlUnit" "G120ControlUnit", and "S1202ControlUnit" as its subtypes. Therefore, four rule function blocks, the super-type and one for each subtype, were created for the execution of this model transformation rule:

- "rule_cu2control_DriveControlUnit"
- "rule_cu2control_DriveControlUnit_G120ControlUnit"
- "rule_cu2control_DriveControlUnit_S120ControlUnit"
- "rule_cu2control_DriveControlUnit_S1202ControlUnit"

Each rule execution function block handles the matching of the source pattern for one the metamodel object types. These multiple rules are required since the current edition of IEC 61131-3 does not support inheritance as already mentioned within the description of the IEC 61131-3 engineering metamodel.

The call to a generated rule function block already fixes the type context for the source pattern match. As specified in Section 6.4.2, elements in the model instances are not dynamically created but active or inactive. Therefore, the first part of a source pattern match checks if the referenced source model element is active. If no source pattern filter is defined, the source pattern matches each active source model element. The rule "rule_do2to_DriveObject" includes an additional filter, which uses a user defined function "MatchString" to check the name of the source model instances for the substring "frontLabel".

For a successful source pattern match, the target model elements are created by the factory functions available for the model instances, for example the "createTechnologyObject" function to create a "TechnologyObject" instance. Finally, again with the help of user defined functions, the bindings for the attributes of the target model elements are assigned.

The implementation of the PIM-MT to real-time PSM-MT higher order transformation is listed in Section 9.6.

```
// SubTypes Sequence {}
FUNCTION_BLOCK rule_do2to_DriveObject
...
    IF DriveObjectInstances[iterator].active THEN
      // filter match
      IF
MatchString(DriveObjectInstances[iterator].name,
'frontLabel') THEN
          sourceMatch:=TRUE;
      END_IF;

    END_IF;
    IF sourceMatch THEN
        // Binding
        // create target element type TechnologyObject
targetIdentifier:=RDTechnologyObject(DriveObjectInstance
s[iterator].name);
targetIterator:=createTechnologyObject(targetIdentifier)
;
        IF (targetIterator>0) THEN
            // set additional attributes
TechnologyObjectInstances[targetIterator].name:=RDTechno
logyObject(DriveObjectInstances[iterator].name); //
HelperCall();

TechnologyObjectInstances[targetIterator].address:=Drive
ObjectInstances[iterator].address;
        ELSE
            DebugMessage(message:='Failed creating ',
parameter:=targetIdentifier);
        END_IF;
    END_IF;
END_FUNCTION_BLOCK
// SubTypes Sequence {'S120ControlUnit',
'G120ControlUnit', 'S1202ControlUnit'}
FUNCTION_BLOCK rule_cu2control_DriveControlUnit
...
END_FUNCTION_BLOCK

FUNCTION_BLOCK
rule_cu2control_DriveControlUnit_G120ControlUnit
...
END_FUNCTION_BLOCK
```

Figure 88: IEC 61131-3 Platform Specific Model Transformation Rule

6.4.4 Rule Execution

The structure of the library, which holds the IEC 61131 platform specific metamodel specification, model instances, and rule execution function blocks, is independent of the program structure and the machine application, for example the bottle labeling machine considered as an example within this thesis. Most parts of this model transformation execution engine can be generated from abstract engineering model definitions.

The integration of the model transformation rules into the execution system of an automation controller is a manual programming task. The time slots, which can be used for the execution of the model transformation rules, depend on the automation task: for some machines, the model transformation rules can be run continuously in any operating state of the machine, for other machines, the model transformation rules can be run only in standby or manual operation mode.

For the bottle labeling machine example, the model transformation rules can be executed as part of the starting state of the machine state machine as already mentioned in Section 4.3.4. Therefore, a program block “RuleExecution” was developed, which can be executed as part of a motion task within the starting state of the bottle labeling machine. As shown in Figure 89, this program block subsequently executes all model transformation until all source model elements have been checked. The program block “RuleExecution” is run after a program block “ModelInitialization”. The program block “ModelShutdown” stops the model transformation engine.

For the real-time model transformation engine developed as part of this thesis, the termination of the rule execution is not influenced by element creation, because no elements are dynamically created but are always available, either in active or inactive state. Therefore, the complete processing of all model elements is not influenced by the creation of new model elements but always relies on a fixed number of model elements available.

```
PROGRAM RuleExecution
    VAR
        rule_do2to_DriveObject :
MT.rule_do2to_DriveObject;
        rule_cu2control_DriveControlUnit :
MT.rule_cu2control_DriveControlUnit;
        modelCompleted : BOOL:=FALSE;

        executionCount:DINT:=0;
    END_VAR

    DebugMessage(message:='Run rule execution');

    executionCount:=0;
    REPEAT
        DebugMessage(message:='rule_do2to_DriveObject
execution nr. ',
parameter:=DINT_TO_STRING(executionCount));

rule_do2to_DriveObject(modelCompleted=>modelCompleted);
        executionCount:=executionCount+1;
    UNTIL modelCompleted END_REPEAT;

    executionCount:=0;
    REPEAT

DebugMessage(message:='rule_cu2control_DriveControlUnit
execution nr. ',
parameter:=DINT_TO_STRING(executionCount));

rule_cu2control_DriveControlUnit(modelCompleted=>modelCo
mpleted);
        executionCount:=executionCount+1;
    UNTIL modelCompleted END_REPEAT;
...
    DebugMessage(message:='Stopped rule execution');

END_PROGRAM
```

Figure 89: IEC 61131-3 Platform Specific Rule Execution

6.4.5 Summary

The complete environment used for the transformation of a platform independent model transformation specification to a real-time platform specific model transformation specification is summarized in Figure 90. The IEC 61131-3 function blocks specifying the engineering metamodels in structured text (ST) are transformed by an XTEXT parser to an Ecore model representing the abstract syntax tree of the metamodel elements. This Ecore model is transformed to a platform independent Ecore metamodel by an ATL transformation.

The platform independent metamodel is used by the platform independent model transformation specification, which defines the transformation rules required for the reconfiguration of a bottle labeling machine. This platform independent rule specification is transformed by an ATL higher order transformation to a real-time platform specific rule execution engine, which is executed on a SIMOTION D435 programmable logic controller.

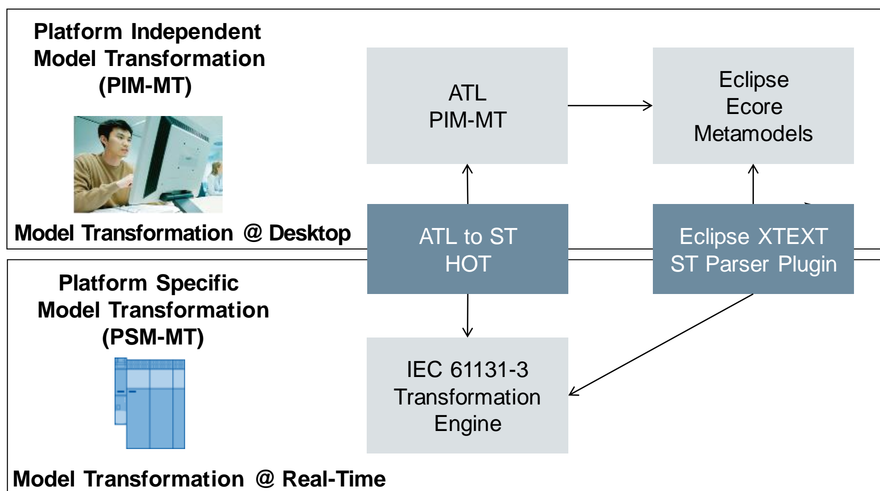

Figure 90: Real-Time PIM-MT to PSM-MT Higher Order Transformation (HOT)

7 Conclusion and Future Work

Machine engineering, like the engineering of a production machine as presented for a bottle labeling machine in Section 2 with its multi-disciplinary engineering models is predestinated for the application of model transformation technology. Depending on the application scenario and the machine builders' engineering environment, the execution of engineering model transformations is not limited to a single execution platform, but must be supported in different execution environments.

Therefore, Section 3 provided a new classification of engineering model transformations by desktop model transformations, enterprise model transformations, and real-time model transformations. Current model transformation classifications are based on characteristics of the model transformation technologies used and not on the view of the application requirements. As shown in the application scenario, these model transformation execution environments are not used alternatively for each new model transformation specification, but the same model transformation specification shall be executed on different execution environments depending on the machine configuration and the engineering workflow. The solution proposed by this thesis for this requirement is a new application of the model driven architecture (MDA): the transformation of platform independent model transformations (PIM-MT) to platform specific model transformations (PSM-MT) by a higher order transformation (HOT).

As a prerequisite for the application of PIM-MT to PSM-MT transformations, Section 3 analyzes the common characteristics of engineering metamodels used in machine development and the common characteristics of model transformation specifications to reconcile these engineering models. Metamodels used up to now by model transformation specification are intended to provide a very detailed object oriented model of a single system with specifications of features like types and references. For engineering metamodels applied to many engineering domain, a simple metamodel was provided in Section 3 with weak references and simplified type handling for attributes, which can be used across the engineering disciplines required by machine engineering. Together with the engineering metamodel also a simplified model transformation specification metamodel was introduced, which is applicable across multiple platform specific model transformation execution engines.

The three different specific platforms for the execution of engineering model transformations are presented in detail in Section 4. For each platform, the rule language, the system model, the pattern language, the inter-rule execution control, and the modularization features are presented in detail. Desktop model transformations, as the key area of academic research, match the required features well and can be used for engineering model transformations in general without any extensions. The XML technology used for enterprise model transformations is more difficult to adapt to model transformation specifications. Therefore, a mapping of the model transformation specification metamodel to the features of XSLT 2.0 was developed. Finally, the execution of model transformations on real-time systems as required by the application

scenario was not considered before this thesis. For IEC 61131-3 programmable logic controllers, a real-time model transformation engine was developed as part of the work of this thesis. The new IEC 61131-3 model transformation engine provides an adaptation of the object oriented engineering metamodels and model transformation specification to the structured programming model of IEC 61131-3.

After the presentation of the platform specific model transformation engines, Section 5 introduces the platform independent model transformation language specified by this thesis. With industrial usage in mind, this thesis did not take the usual approach of defining a new platform independent model transformation language, but proposes the new approach of reusing an existing model transformation specification language as a platform independent model transformation specification. Using an existing model transformation language allows to build on an established language specification and a proven tool environment. To be usable as a platform independent model transformation, the use of language features of the chosen ATL model transformation language must be restricted. This affects features used in meta-modeling like the already mentioned usage of references as well as features of the rule language like the pattern definition. Otherwise, a PIM-MT to PSM-MT transformation could not be implemented as a higher order model transformation but compiler technology would be required. For the transformation of the pattern language, the application of user defined functions on the platform independent level and the platform specific level was developed as a new concept within this thesis, which showed its usefulness for keeping transformation rules simple but still allowing complex pattern definitions.

With respect to the PIM-MT to PSM-MT higher order transformation, this thesis revealed that this transformation always requires a reverse directed transformation of the platform specific metamodel (PSM-MM) to a platform independent metamodel (PIM-MM) used by the platform independent model transformation specification. While the PIM-MT to PSM-MT higher order transformation is a one to many transformation (a single platform independent metamodel specification is transformed to multiple platform specific metamodel transformations), the PSM-MM to PIM-MM transformation is a many to one transformation (multiple platform specific metamodels are transformed to a single platform independent metamodel).

Finally, Section 6 presents the implementation of the PIM-MT to PSM-MT higher order transformations together with the PSM-MM to PIM-MM transformations for the three platform specific model transformation environments as an evaluation of the work of this thesis. For desktop model transformations, ATL was used as the implementation. Enterprise model transformations were implemented as part of the Siemens Teamcenter PLM environment. The Siemens SIMOTION programmable logic controllers were used for the evaluation of real time controllers.

Providing a new approach for an existing challenge, like the PSM-MT to PIM-MT approach for the platform independent specification of model transformation answers a set of open issued but also opens new questions for further research.

The platform independent model transformation specification used within this thesis relies on a set of implicit assumptions, which must be met by the platform specific model transformation implementation. For the application of model transformations to already existing models (which is the standard case for engineering model transformation) a specification for the deletion of objects is required. For engineering models including objects with a dependency to the outside world of the model transformation system (which is common for enterprise model transformations and real-time model transformations) a definition of the rule execution order is required. It is an open question, what other language features are required for a model transformation language if the requirement is not to handle a single system in all details but to provide a transformation language that can be cover the execution requirements of many different model transformation platforms.

Another important work of this thesis was the simplification of the model transformation specification and the metamodel definitions to move away from the inflexible and strict handling of references to widen the applicability of the specification to different execution system. Within the work of this thesis, this was for example achieved by implementing reference handling and object identity based on strings generated by user defined functions instead of meta-modeling concepts. Although gaining flexibility, it would be desired still to support checks, if references to object identifiers are valid, for example with respect to the referenced object type or with respect to the structure of the object identifier. Within database modeling, it is common to specify constraints on table columns, e.g. columns serving as primary key (a reference in the wording of object oriented design) or for valid values of database columns. It is a direction of future research, if model transformation languages could be extended by these database modeling concepts to support the applications considered by this thesis.

Finally, the higher order transformation from the PIM-MT to the PSM-MT is currently implemented as a desktop model transformation based on the ATL model transformation language. The application of the concepts provided by this thesis would also allow the specification of this higher order transformation on a platform independent level and the execution of this higher order transformation on different execution platforms. The application fields and the requirements of such a successive application of the PIM-MT to PSM-MT principle are not investigated yet.

8 References

[Am10] American National Standards Institute (ANSI). ANSI/ISA 88.00.01: Batch Control Part 1: Models and Terminology, 2010.

[AT05] ATLAS group LINA & INRIA: The ATL to Problem ATL transformation, 2005. http://www.eclipse.org/atl/atlTransformations/#ATL2Problem, 16.05.2013.

[BK05] Bézivin, J.; Kurtev, I.: Model-based Technology Integration with the Technical Space Concept. In (Nürnberg, P. Ed.): MIS '05: Proceedings of the 2005 symposia on Metainformatics. Esbjerg, Denmark. ACM, 2005.

[Bo04] Booch, G. et al.: An MDA Manifesto. In (Frankel, D. S.; Parodi, J. Ed.): The MDA Journal. Meghan Kiffer Printing, November 30, 2004; pp. 1–9.

[CH03] Czarnecki, K.; Helsen, S.: Classification of Model Transformation Approaches. In (Bettin, J. Ed.): 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. Anaheim, 2003; pp. 1–17.

[DHI12] Doan, A.; Halevy, A.; Ives, Z. G.: Principles of data integration. Elsevier Morgan Kaufmann, Amsterdam, 2012.

[DM07] Dunphy, G.; Metwally, A.: Pro BizTalk 2006. George Dunphy Ahmed Metwally, Berkeley, CA, 2007.

[DW09] Dawson, J.; Wainwright, J.: Pro Mapping in BizTalk Server 2009. Apress, Berkeley, CA, 2009.

[ES09] Eigner, M.; Stelzer, R.: Product Lifecycle Management. Ein Leitfaden für Product Development und Life Cycle Management. Springer, Berlin, 2009.

[Fo82] Forgy, C. L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. In Artificial Intelligence, 1982, 19; pp. 17–37.

[FR07] France, R. B.; Rumpe, B.: Model-driven development of complex software: A research roadmap. In (Briand, L. C.; Wolf, A. L. Ed.): Future of software engineering, 2007. FOSE '07 ; 23 - 25 May 2007, Minneapolis, Minnesota ; [at] ICSE 2007, [29th International Conference on Software Engineering]. IEEE Computer Society, Los Alamitos, Calif., 2007; pp. 37–54.

[FT11] Frey, G.; Thramboulidis, K.: Einbindung der IEC 61131 in modellgetriebene Entwicklungsprozesse: Automation 2011. Der Automatisierungskongress in Deutschland ; Kongress Baden-Baden, 28. und 29. Juni 2011; 12. Branchentreff der Mess- und Automatisierungstechnik. VDI-Verl., Düsseldorf, 2011; pp. 21–24.

[Ge12] Geus, T.: Syntaxeditor für speicherprogrammierbare Steuerungen auf Basis des Xtext Frameworks. Masterarbeit, Coburg, 2012.

[Ho12] Hofer, J.: SCL und OOP mit dem TIA Portal V11. Ein Leitfaden für eine objektorientierte Arbeitsweise. VDE Verl., Berlin, 2012.

[In03a] International Electrotechnical Commission (IEC). IEC 61131-1: Programmable controllers – Part 1: General information, 2003a.

[In03b] International Electrotechnical Commission (IEC). IEC 61131-3: Programmable controllers – Part 3: Programming languages, 2003b.

[In12] International Electrotechnical Commission (IEC). IEC 61131-3: Programmable controllers – Part 3: Programming languages, 2012.

[IS05] ISO/IEC JTC 1/SC 32 Data Management and Interchange. ISO/IEC 19503: Information technology - XML Metadata Interchange (XMI), 2005.

[IS09] ISO/TC 10 Technical product documentation. IEC 81346-1: Industrial systems, installations and equipment and industrial products - Structuring principles and reference designations - Part 1: Basic rules, 2009.

[IS11] ISO/IEC JTC 1/SC 32 Data Management and Interchange. ISO/IEC 9075-1: Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework), 2011.

[IS12a] ISO/IEC JTC 1/SC 7 Software and Systems Engineering. ISO/IEC 19505-1: Information technology - Object Management Group Unified Modeling Language (OMG UML) - Part 1: Infrastructure, 2012a.

[IS12b] ISO/IEC JTC 1 Information Technology Standards. ISO/IEC 19507: Information technology - Object Management Group Object Constraint Language (OCL), 2012b.

[Jé05] Jézéquel, J.-M.: A MDA Approach to Model & Implement Transformations. In (Bézivin, J.; Heckel, R. Ed.): Language Engineering for Model-Driven Software Development, 2005; pp. 1–5.

[JK07] Jouault, F.; Kurtev, I.: On the interoperability of model-to-model transformation languages. In Science of Computer Programming, 2007, 68; pp. 114–137.

[Jo06] Jouault, F.: Contribution à l’étude des langages de transformation de modèles, 2006.

[KBA02] Kurtev, I.; Bézivin, J.; Aksit, M.: Technological Spaces: an Initial Appraisal. CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine.: CoopIS, DOA'2002 Federated Conferences, Industrial track. Irvine, October 28-November 1, 2002., 2002; pp. 1–6.

[Ko13] Kolovos, D. S. et al.: The epsilon Book, 2013. http://www.eclipse.org/gmt/epsilon/doc/book/, 22.02.2013.

[KPP06] Kolovos, D. S.; Paige, R. F.; Polack, F. A. C.: The Epsilon Object Language (EOL). In (Rensink, A. Ed.): Model driven architecture - foundations and applications. Second European conference, ECMDA-FA 2006, Bilbao, Spain, July 10 - 13, 2006 ; proceedings. Springer, Berlin, 2006; pp. 128–142.

[KQE11] Krausser, T.; Quirós, G.; Epple, U.: An IEC-61131-based Rule System for Integrated Automation Engineering: Concept and Case Study: Proceedings of the 9th International Conference on Industrial Informatics (INDIN 2011), 2011; pp. 539–544.

[KWB03] Kleppe, A.; Warmer, J.; Bast, W.: MDA explained. The model driven architecture practice and promise. Addison-Wesley, Boston, 2003.

[La04] Lambers, L.: A New Version of GTXL: An Exchange Format for Graph Transformation Systems. In (Tom Mens; Andy Schürr; Gabriele Taentzer Ed.): Proc. Workshop on Graph-Based Tools (GraBaTs‘04), Satellite Event of ICGT‘04. Elsevier Science, Rom, Italy, 2004; pp. 51-63.

[Li00] Linthicum, D. S.: Enterprise application integration. Addison-Wesley, Boston, 2000.

[LS06] Lawley, M.; Steel, J.: Practical Declarative Model Transformation with Tefkat. In (Bruel, J.-M. Ed.): Satellite events at the MoDELS 2005 conference. MoDELS 2005 international workshops ; doctoral symposium, educators symposium ; Montego Bay, Jamaica, October 2 - 7, 2005 ; revised selected papers. Springer, Berlin, 2006; pp. 139–150.

[Lu11] Lui, M. et al.: Pro Spring Integration. Apress, Berkeley, CA, 2011.

[Ma08] Maurmaier, M.: Modell-zu-Modell-Transformationen in der Automatisierungstechnik: Automation 2008. Lösungen für die Zukunft. VDI Berichte 2032. VDI-Verl., Düsseldorf, 2008; pp. 231–234.

[Mi06] Mirkheshti, R.: Zur Entwicklung von Berechnungsmethoden und deren Integration in den Produktentwicklungsprozess. Dissertation. Fraunhofer IRB-Verl., Berlin, 2006.

[MM03] Mukerji, J.; Miller, J.: MDA Guide Version 1.0.1, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf, 03.05.2013.

[MV05] Mens, T.; Van Gorp, P.: A Taxonomy of Model Transformation and its Application to Graph Transformation. In (Karsai, G.; Taentzer, G. Ed.): Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). Tallinn, Estonia, September 28, 2005, 2005; pp. 1–27.

[Ob10] Object Management Group: Object Constraint Language (OCL), Version 2.2, 2010. http://www.omg.org/spec/OCL/2.2/PDF, 29.06.2010.

[Ob11] Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2011. http://www.omg.org/spec/QVT/1.1/, 17.05.2013.

[Or06] Organization for Machine Automation and Control (OMAC): Guidelines for Packaging Machinery Automation V3.1, 2006. http://www.omac.org/filestore/omac/OPW_Guidelines_v3_1_Official.pdf, 25.01.2010.

[RJB05] Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language reference manual. Addison-Wesley, Boston, 2005.

[Sc95] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In (Mayr, E. W.; Schmidt, G.; Tinhofer, G. Ed.): Graph-Theoretic Concepts in Computer Science. 20th International Workshop, WG '94 Herrsching, Germany, June 1618, 1994 Proceedings. Springer, Berlin, Heidelberg, 1995; pp. 151–163.

[Si08a] Siemens AG: Motion Control System SIMOTION, 2008. http://www.siemens.com/simotion, 09.10.2008.

[SI08b] Saaksvuori, A.; Immonen, A.: Product Lifecycle Management. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2008b.

[Si11a] Siemens AG: Open product lifecycle data sharing using XML, 2011. http://www.plmxml.com, 26.03.2013.

[Si11b] Siemens AG: Programmable Logic Controllers SIMATIC, 2011. http://www.siemens.com/simatic, 28.11.2011.

[SK12] Schlereth, M.; Krausser, T.: Platform-Independent Specification of Model Transformations @ Runtime Using Higher-Order Transformations. In (Sinz, E. J.; Schürr, A. Hrsg.): Modellierung 2012. 14. - 16. März 2012, Bamberg. Ges. für Informatik, Bonn, 2012; pp. 123–138.

[St09] Steinberg, D. et al.: EMF - Eclipse modeling framework. Addison-Wesley, Upper Saddle River, NJ, 2009.

[Te05] Tennison, J.: Beginning XSLT 2.0. From Novice to Professional. Springer, Dordrecht, 2005.

[Ti09] Tisi, M. et al.: On the Use of Higher-Order Model Transformations. In (Paige, R. F.; Hartman, A.; Rensink, A. Ed.): Model driven architecture - foundations and applications. 5th European conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23 - 26, 2009. Springer, Berlin, 2009; pp. 18–33.

[VD04] VDI-Gesellschaft Entwicklung Konstruktion Vertrieb. VDI 2206: Entwicklungsmethodik für mechatronische Systeme, 2004.

[VW07] Vogel-Heuser, B.; Wannagat, A.: Wiederverwendung und Modulares Engineering mit CoDeSys 3.0. Für Automatisierungslösungen mit objektorientiertem Ansatz. Oldenbourg Industrieverlag; Oldenbourg-Industrieverl., München, 2007.

[Wi03] Willink, E. D.: UMLX : A graphical transformation language for MDA. In (Bettin, J. Ed.): 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. Anaheim, 2003; pp. 1–12.

[Wi12] Wimmer, M. et al.: A Catalogue of Refactorings for Model-to-Model Transformations. In Journal of Object Technology, 2012, 11; pp. 2:1-40.

[Wo07] World Wide Web Consortium (W3C): XSL Transformations (XSLT) Version 2.0, 2007.

[Wo08] World Wide Web Consortium (W3C): Extensible Markup Language (XML) 1.0, 2008.

[Wo10] World Wide Web Consortium (W3C): XML Path Language (XPath) 2.0 (Second Edition), 2010.

[Wo12] World Wide Web Consortium (W3C): XML Schema Definition Language (XSD) 1.1 Part 1: Structures, 2012.

[YW09] Yie, A.; Wagelaar, D.: Advanced Traceability for ATL. In (Jouault, F. Ed.): 1st International Workshop on Model Transformation with ATL. Co-located with the 10th Libre Software Meeting, Nantes, France, July 8-9, 2009, 2009; pp. 78–87.

[ZP08] Zäh, M.; Pörnbacher, C.: Model-driven development of PLC software for machine tools. In Production Engineering, 2008.

[ZSW99] Zündorf, A.; Schürr, A.; Winter, A. J.: Story Driven Modeling. Technical Report tr-ri-99-211, 1999.

9 Appendix

The appendix lists the implementations of the model transformations implemented as part of the evaluation presented in Section 6.

9.1 PIM-MT to intermediate ATL transformation

```
-- @path MMPIMMT=/pim-hot/model/pimmt.ecore
-- @nsURI MMATL=http://www.eclipse.org/gmt/2005/ATL

module atl2pimmt;
create OUTPIMMT : MMPIMMT from INATL : MMATL;

rule module2module
{
	from
		s: MMATL!Module
	to
		t: MMPIMMT!Module
		(
			elements <- s.elements->select(e |
e.oclIsTypeOf(MMATL!MatchedRule))
			, name <- s.name
		)
}

rule matchedRule2rule
{
	from
		s: MMATL!MatchedRule
	to
		t: MMPIMMT!Rule
		(
			name <- s.name
			, inPattern <- s.inPattern.elements.at(1)
			-- , filter <-
thisModule.operationCallExp2helperCall(s.inPattern.filter)
			-- , filter <- s.inPattern.filter
			, outPatterns <- s.outPattern.elements


		)
}
```

```
rule simpleOutPatternElement2outPattern
{
	from
		s: MMATL!OutPatternElement
	to
		t: MMPIMMT!OutPattern
		(
			type <- s.type.name
			, bindings <- s.bindings
		)
}

rule binding2binding
{
	from
		s: MMATL!Binding
	to
		t: MMPIMMT!Binding
		(
			propertyName <- s.propertyName
			, value <- s.value
			-- , value <- u
		)
--		, u: MMPIMMT!SimpleExpression
--		(
--			type <- 'String'
--			, value <- 'helloBinding'
--		)
--		, u: MMPIMMT!HelperCall
--		(
--			name <- 'helloHelperCall'
--			, arguments <- v
--		)
--		, v: MMPIMMT!SimpleExpression
--		(
--			type <- 'String'
--			, value <- 'helloArgument'
--		)

}
```

```
-- InPattern

rule simpleInPatternElement2inPattern
{
	from
		s: MMATL!SimpleInPatternElement
		(
			s.oclIsKindOf(MMATL!SimpleInPatternElement)
		)
	to
		t: MMPIMMT!InPattern
		(
			type <- s.type.name
			, filter <- s.refImmediateComposite().filter
		)

}

rule operationCallExp2helperCall
{
	from
		s: MMATL!OperationCallExp
		(
			true --
s.refImmediateComposite().oclIsKindOf(MMATL!InPattern)
		)
	to
		t: MMPIMMT!HelperCall
		(
			name <- s.operationName
			, arguments <- s.arguments->select(e |
e.oclIsTypeOf(MMATL!NavigationOrAttributeCallExp) or
e.oclIsTypeOf(MMATL!StringExp))
		)
}
```

```
-- test: support a.b and a.b.c attributes
rule navigationOrAttributeCallExp2attributeArgument
{
	from
		s: MMATL!NavigationOrAttributeCallExp
		(
			true --
s.refImmediateComposite().refImmediateComposite().oclIsKindOf(MM
ATL!InPattern)
		)
	to
		t: MMPIMMT!SimpleExpression
		(
			type <- 'Attribute'
			, value <- if
(s.source.oclIsTypeOf(MMATL!NavigationOrAttributeCallExp)) then
s.source.name+'.'+s.name else s.name endif

		)


}

rule stringExp2stringArgument
{
	from
		s: MMATL!StringExp
		(
			true --
s.refImmediateComposite().refImmediateComposite().oclIsKindOf(MM
ATL!InPattern)
		)
	to
		t: MMPIMMT!SimpleExpression
		(
			type <- 'String'
			, value <- s.stringSymbol

		)


}
```

```
lazy rule lzoperationCallExp2helperCall
{
	from
		s: MMATL!OperationCallExp
	to
		t: MMPIMMT!HelperCall
		(
			name <- s.operationName
		)
}
```

9.2 Desktop PIM-MT/Intermediate Representation to PSM-MT transformation

```
-- @path MMPIMMT=/pim-hot/model/pimmt.ecore

query pimmt2atl = MMPIMMT!Module.allInstances()-
>first().toString().writeTo(
       '/pim-hot/model-gen/'+MMPIMMT!Module.allInstances()-
>first().name+'.psmmt.atl');

helper context MMPIMMT!Module def: toString() : String =
'-- @path MMELECTRICAL=/pim_mt/model/electrical.ecore\n' +
'-- @path MMAUTOMATION=/pim_mt/model/automation.ecore\n' +
'\n' +
'module electrical2automation;\n\n' +
'create OUTAUTOMATION : MMAUTOMATION from INELECTRICAL :
MMELECTRICAL;\n\n' +
'uses userDefinedFunctions;\n\n' +

       self.elements->iterate(e; acc : String = '' | acc +
e.toString() + '\n');

helper context MMPIMMT!Rule def: toString() : String =
'rule '+self.name+'\n' +
'{\n' +
'  from\n' +
             self.inPattern.toString() +
'  to\n' +
       self.outPatterns->iterate(e; acc : String = '' | acc +
e.toString(self.outPatterns.indexOf(e)) + if
e=self.outPatterns.last() then '\n' else ',\n' endif + '\n') +
'      do\n' +
'      {\n' +
'             thisModule.debug(\''+self.name+'\');\n' +
'      }             \n' +
'\n' +
'}\n';
```

```
helper context MMPIMMT!InPattern def: toString() : String =
'       s: MMELECTRICAL!'+self.type + '\n' +
       if self.filter->oclIsUndefined() then
'         -- no filter defined\n'
       else
'         ('+
'         -- filter match\n' +
'            '+self.filter.toString()+'\n' +
'         )\n'
       endif;

helper context MMPIMMT!SimpleExpression def: toString() : String
=
       if self.type='String' then
              '\''+self.value+'\''
       else
              's.'+self.value
       endif;

helper context MMPIMMT!HelperCall def: toString() : String =
'thisModule.'+self.name+'(' +
       self.arguments->iterate(e; acc : String = '' | acc +
e.toString() + if e=self.arguments.last() then '' else ', '
endif) +
')';


helper context MMPIMMT!OutPattern def: toString(index : Integer)
: String =
'         t'+index+': MMAUTOMATION!'+self.type+'\n'+
'          (\n'+
              self.bindings->iterate(e; acc : String = '' | acc +
e.toString() + if e=self.bindings.last() then '\n' else ',\n'
endif) +
'          )';


helper context MMPIMMT!Binding def: toString() : String =
'                    '+self.propertyName+' <-
'+self.value.toString();
```

9.3 Enterprise PIM-MT/Intermediate Representation to PSM-MT transformation

```
-- @path MMPIMMT=/pim-hot/model/pimmt.ecore

query pimmt2xsl = MMPIMMT!Module.allInstances()-
>first().toString().writeTo(
       '/pim-hot/model-gen/'+MMPIMMT!Module.allInstances()-
>first().name+'.psmmt.xsl');

helper context MMPIMMT!Module def: toString() : String =
'<?xml version="1.0" encoding="UTF-8"?>\n' +
'<!-- rule module '+self.name +' -->\n'+
'<xsl:stylesheet version="1.0" \n' +
'      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"\n' +
'      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" \n'
+
'      \n' +
'      xmlns:plmxml="http://www.plmxml.org/Schemas/PLMXMLSchema"
\n' +
'      xmlns:xhtml="http://www.w3.org/1999/xhtml" \n' +
'
       xsi:schemaLocation="http://www.plmxml.org/Schemas/PLMXMLSc
hema PLMXMLElectricalSchema.xsd PLMXMLAutomationSchema.xsd "\n'
+
'      \n' +
'      >\n' +
'\n' +
'<xsl:import-schema
namespace="http://www.plmxml.org/Schemas/PLMXMLSchema" \n' +
'                     schema-
location="../model/PLMXMLElectricalSchema.xsd " />  \n' +
'<xsl:import-schema
namespace="http://www.plmxml.org/Schemas/PLMXMLSchema" \n' +
'                     schema-
location="../model/PLMXMLAutomationSchema.xsd " />  \n' +
'      \n' +
'<xsl:output method="xml" indent="yes" />\n' +
'\n' +
'<xsl:template match="/*">\n' +
'      \n' +
```

```
'      <plmxml:PLMXML \n' +
'             author="" \n' +
'             date="2001-01-01" \n' +
'             schemaVersion="0.0" \n' +
'             time="12:00:00" \n' +
'
       xmlns:plmxml="http://www.plmxml.org/Schemas/PLMXMLSchema"
\n' +
'             xmlns:xhtml="http://www.w3.org/1999/xhtml" \n' +
'             xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" \n' +
'
       xsi:schemaLocation="http://www.plmxml.org/Schemas/PLMXMLSc
hema ../model/PLMXMLAutomationSchema.xsd "\n' +
'             >\n' +
'                    \n' +
'             <xsl:apply-templates/>\n' +
'             \n' +
'      </plmxml:PLMXML>\n' +
'</xsl:template>\n' +
'\n' +
'<xsl:include href="userdefinedfunctions.xsl"/>\n'+
       self.elements->iterate(e; acc : String = '' | acc +
e.toString() + '\n') +
'</xsl:stylesheet>\n';

helper context MMPIMMT!Rule def: toString() : String =
'   <!-- rule '+self.name+'-->\n' +
'   <xsl:template match="element(*,
plmxml:'+self.inPattern.type+')'+self.inPattern.toString()+'">\n
' +
       self.outPatterns->iterate(e; acc : String = '' | acc +
e.toString() + '\n') +
'   </xsl:template>\n';

helper context MMPIMMT!InPattern def: toString() : String =
       if self.filter->oclIsUndefined() then
             ''
       else
'['+self.filter.toString()+']'
       endif;
```

```
helper context MMPIMMT!SimpleExpression def: toString() : String
=
	if self.type='String' then
		'\''+self.value+'\''
	else
		'@'+self.value
	endif;

helper context MMPIMMT!HelperCall def: toString() : String =
	'plmxml:'+self.name+
	'(' +
		self.arguments->iterate(e; acc : String = '' | acc +
e.toString() + if e=self.arguments.last() then '' else ', '
endif) +
	')';


helper context MMPIMMT!OutPattern def: toString() : String =
'	<!-- OutPattern -->\n' +
'	<xsl:element name="plmxml:'+self.type+'"
type="plmxml:'+self.type+'">\n' +
		self.bindings->iterate(e; acc : String = '' | acc +
e.toString()) +
'	</xsl:element>\n';


helper context MMPIMMT!Binding def: toString() : String =
'			<xsl:attribute
name="'+self.propertyName+'"><xsl:value-of\n' +
'					select="'+self.value.toString()+'"
/></xsl:attribute>\n';
```

9.4 Real-Time PSM-MM to PIM-MM transformation

```
-- @nsURI MMSTUMC=http://www.xtext.org/iec61131/stumc/StUmc
-- @nsURI MMECORE=http://www.eclipse.org/emf/2002/Ecore

module stumc2ecore;
create OUTECORE : MMECORE from INSTUMC : MMSTUMC;

-- MMSTUMC!Bool_Spec
-- MMSTUMC!String_Spec
helper context MMSTUMC!Var_Init_Decl def : getEDataType() :
MMECORE!EDataType =
	if self.combined_Spec.oclIsKindOf(MMSTUMC!Bool_Spec) then
		MMECORE!EBoolean
	else
		MMECORE!EString
	endif;

rule unit2package
{
	from
		s: MMSTUMC!Unit
	to
		t: MMECORE!EPackage
		(
			name <- s.unitDef.id
			, nsPrefix <- 'org.mtmda.'+ s.unitDef.id
			, nsURI <- 'http://'+s.unitDef.id+'mtmda.org'
			, eClassifiers <- s.implementation.pous
		)

}
rule fb2eClass
{
	from
		s: MMSTUMC!FB_Decl
	to
		t: MMECORE!EClass
		(
			name <- s.fbName
			, eStructuralFeatures <-
s.var_Decls.first().var_Decl
		)
}
```

```
rule varInitDecl2eAttribute
{
	from
		s: MMSTUMC!Var_Init_Decl
		(
	s.refImmediateComposite().oclIsKindOf(MMSTUMC!Input_Decls)
		)
	to
		t: MMECORE!EAttribute
		(
			name <-
s.variable_List.variable_Name.first().name
			, eType <- s.getEDataType()
		)
}
```

9.5 Real-Time PIM-MM generalizations transformation

```
-- @nsURI MMECORE=http://www.eclipse.org/emf/2002/Ecore

module sttypes2supertypes;
create OUTSTTYPES : MMECORE refining INSTTYES : MMECORE;

rule sElModel
{
	from
		s:MMECORE!EClass
		(
			s.name='ElModel'
		)
	to
		t:MMECORE!EClass
		(
			eSuperTypes <- MMECORE!EClass.allInstances()-
>select(e | e.name='ElClass')
		)
}

rule sDriveControlUnit
{
	from
		s:MMECORE!EClass
		(
			s.name='DriveControlUnit'
		)
	to
		t:MMECORE!EClass
		(
			eSuperTypes <- MMECORE!EClass.allInstances()-
>select(e | e.name='ElClass')
		)
}
```

```
rule sDriveObject
{
	from
		s:MMECORE!EClass
		(
			s.name='DriveObject'
		)
	to
		t:MMECORE!EClass
		(
			eSuperTypes <- MMECORE!EClass.allInstances()-
>select(e | e.name='ElClass')
		)
}

rule sG120ControlUnit
{
	from
		s:MMECORE!EClass
		(
			s.name='G120ControlUnit'
		)
	to
		t:MMECORE!EClass
		(
			eSuperTypes <- MMECORE!EClass.allInstances()-
>select(e | e.name='DriveControlUnit')
		)
}

rule sS120ControlUnit
{
	from
		s:MMECORE!EClass
		(
			s.name='S120ControlUnit'
		)
	to
		t:MMECORE!EClass
		(
			eSuperTypes <- MMECORE!EClass.allInstances()-
>select(e | e.name='DriveControlUnit')
		)
}
```

```
rule sS1202ControlUnit
{
	from
		s:MMECORE!EClass
		(
			s.name='S1202ControlUnit'
		)
	to
		t:MMECORE!EClass
		(
			eSuperTypes <- MMECORE!EClass.allInstances()-
>select(e | e.name='S120ControlUnit')
		)
}
```

9.6 Real-Time PIM-MT/Intermediate Representation to PSM-MT transformation

```
-- @path MMPIMMT=/pim-hot/model/pimmt.ecore

-- query pimmt2st = MMPIMMT!Module.allInstances()-
>first().toString().writeTo('/pim-hot/model-gen/psmmt.stumc');
query pimmt2st = MMPIMMT!Module.allInstances()-
>first().toString().writeTo(
	'/pim-hot/model-gen/'+MMPIMMT!Module.allInstances()-
>first().name+'.psmmt.stumc');
-- ->collect(e | e.toString().writeTo('/pim-hot/model-
gen/psmmt.stumc'));

helper def : getDomainType(type : String) : MMECORE!EClass =
	MMECORE!EClass.allInstances()->select(e |
e.name=type).first();

helper def : getSubTypes(type : String) : Sequence(String) =
	let
		eclassInstances : Sequence(MMECORE!EClass) =
MMECORE!EClass.allInstances().debug('eclassInstances')
	in let
		domainType : MMECORE!EClass =
thisModule.getDomainType(type).debug('domainType')
	in let
		subTypes : Sequence(MMECORE!EClass) =
MMECORE!EClass.allInstances()->select(e |
e.eAllSuperTypes.includes(domainType)).debug('getSubTypes for
'+type)
	in
		if (subTypes.size()>0) then
			subTypes->collect(e | e.name)
		else
			Sequence{}
		endif;
```

```
helper context MMPIMMT!Module def: toString() : String =
'// rule module '+self.name +'\n' +
'INTERFACE\n' +
'     USELIB DebugFile;\n' +
'     // manual configuration of the USES statement is
required\n' +
'     USES electricalInstances, automationInstances,
UserDefinedFunctions;\n' +
'\n' +
'     \n' +
       self.elements->iterate(e; acc : String = '' | acc +
e.ruleInterfaceDeclaration() + '\n') +
'     \n' +
'         \n' +
'END_INTERFACE\n' +
'\n' +
'IMPLEMENTATION\n' +
'          \n' +
       self.elements->iterate(e; acc : String = '' | acc +
e.toString() + '\n') +
'     \n' +
'     \n' +
'     \n' +
'END_IMPLEMENTATION\n\n';

helper context MMPIMMT!Rule def: ruleFbName() : String =
       'rule_'+self.name+'_'+self.inPattern.type;

helper context MMPIMMT!Rule def: ruleInterfaceDeclaration() :
String =
'     FUNCTION_BLOCK '+self.ruleFbName()+';\n' +
       thisModule.getSubTypes(self.inPattern.type)->iterate(e;
acc : String = '' | acc +
'     FUNCTION_BLOCK '+self.ruleFbName()+'_'+e+';\n'

       );


helper context MMPIMMT!Rule def: toString() : String =
' // SubTypes
'+thisModule.getSubTypes(self.inPattern.type).toString()+' \n'+
       self.subRule('') +
       thisModule.getSubTypes(self.inPattern.type)->iterate(e;
acc : String = '' | acc + self.subRule(e))
       ;
```

```
helper context MMPIMMT!Rule def: subRule(inPatternSubType :
String) : String =
	let
		inPatternType : String =
			if (inPatternSubType.size()>0) then
				inPatternSubType
			else
				self.inPattern.type
			endif
	in let
		fb_name : String =
			if (inPatternSubType.size()>0) then
				self.ruleFbName()+'_'+inPatternSubType
			else
				self.ruleFbName()
			endif
	in
		'	FUNCTION_BLOCK '+fb_name+'\n' +
'		VAR_OUTPUT\n' +
'			modelCompleted : BOOL;\n' +
'		END_VAR\n' +
'		\n' +
'		VAR\n' +
'			iterator, targetIterator : DINT :=1;\n' +
'			stateModelCompleted : BOOL :=FALSE;\n' +
'		END_VAR\n' +
'		\n' +
'		VAR_TEMP\n' +
'			sourceMatch:BOOL :=FALSE;\n' +
'			targetIdentifier : STRING;\n' +
'		END_VAR\n' +
'		\n' +
'		DebugMessage(message:=\''+fb_name+' execution, object
name\',
parameter:='+inPatternType+'Instances[iterator].name);\n' +
'		IF '+inPatternType+'Instances[iterator].active THEN\n'+
			self.inPattern.toString(inPatternType) +
'		END_IF;\n'+
'		\n' +
'		IF sourceMatch THEN\n' +
'			DebugMessage(message:=\'sourceMatch=TRUE\');\n' +

				self.outPatterns->iterate(e; acc : String =
'' | acc + e.toString(inPatternType) + '\n') +

'		END_IF;\n' +
```

```
'         \n' +
'         iterator:=iterator+1;\n' +
'         IF
iterator>_lastIndexOf(in:='+inPatternType+'Instances) THEN\n' +
'             iterator:=1;\n' +
'             stateModelCompleted:=TRUE;\n' +
'         END_IF;\n' +
'         \n' +
'         modelCompleted:=stateModelCompleted;    \n' +
'         \n' +
'     END_FUNCTION_BLOCK\n\n';

helper context MMPIMMT!InPattern def: toString(sourceType :
String) : String =
       if self.filter->oclIsUndefined() then
'          // no filter defined\n' +
'          sourceMatch:=TRUE;\n' +
'          \n'
       else
'          // filter match        \n' +
'          IF '+self.filter.toStringSourceType(sourceType)+'
THEN\n' +
'              sourceMatch:=TRUE;\n' +
'          END_IF;\n' +
'          \n'
       endif;

helper context MMPIMMT!SimpleExpression def: toString() : String
=
       '// PSMMT-Error: MMPIMMT!SimpleExpression def: toString()
used without sourceType\n';
helper context MMPIMMT!HelperCall def: toString() : String =
       '// PSMMT-Error: MMPIMMT!HelperCall def: toString() used
without sourceType\n';

helper context MMPIMMT!SimpleExpression def:
toStringSourceType(sourceType : String) : String =
       if self.type='String' then
              '\''+self.value+'\''
       else
              if (sourceType='') then
                     self.value
              else
                     sourceType+'Instances[iterator].'+self.value
              endif
       endif;
```

```
helper context MMPIMMT!HelperCall def:
toStringSourceType(sourceType : String) : String =
	self.name+
	'(' +
		self.arguments->iterate(e; acc : String = '' | acc +
e.toStringSourceType(sourceType) + if e=self.arguments.last()
then '' else ', ' endif) +
	')';


helper context MMPIMMT!OutPattern def: toString(sourceType :
String) : String =
	let
		identifierBinding : MMPIMMT!Binding = self.bindings-
>select(e | e.propertyName='name').first()
	in
'		// Binding\n' +
'		// create target element type '+ self.type + '\n' +
	if identifierBinding.oclIsUndefined() then
'		// PSMMT-Error: no identifier\n'
	else

'
targetIdentifier:='+identifierBinding.value.toStringSourceType(s
ourceType)+';\n' +
'
targetIterator:=create'+self.type+'(targetIdentifier);\n' +

'		IF (targetIterator>0) THEN \n' +
'		    DebugMessage(message:=\'Created \',
parameter:=targetIdentifier);\n' +
'		    // set additional attributes\n' +
		self.bindings->iterate(e; acc : String = '' | acc +
'		    ' +

	self.type+'Instances[targetIterator].'+e.propertyName+':='
+
				if
(e.value.oclIsKindOf(MMPIMMT!SimpleExpression)) then
					if (e.value.type='String')
```

```
then

       '\''+e.value.value+'\''
                                                    else

       sourceType+'Instances[iterator].'+e.value.value
                                                    endif +
                                                    ';\n'
                                             else

       e.value.toStringSourceType(sourceType)+'; //
HelperCall();\n'
                                             endif
                    ) +

'             ELSE\n' +
'                 DebugMessage(message:=\'Failed creating \',
parameter:=targetIdentifier);\n' +
'             END_IF;\n' +
'             \n'
       endif;

helper context MMPIMMT!Binding def: toString() : String =
       '.'+self.propertyName+':=';
```

Curriculum Vitae

Der Lebenslauf ist in der Online-Version aus Gründen des Datenschutzes nicht enthalten.

Veröffentlichungen

Schlereth, Michael; Krausser, Tina (2012): Platform-Independent Specification of Model Transformations @ Runtime Using Higher-Order Transformations. In: Sinz, Elmar J.; Schürr, Andy (Hg.): Modellierung 2012. 14. - 16. März 2012, Bamberg. Bonn: Ges. für Informatik (GI-EditionProceedings, 201), S. 123–138.

Schlereth, Michael; Lauder, Marius; Rose, Sebastian; Schürr, Andy (2010): Concurrent Model Driven Automation Engineering. Building Engineering Tool Integration Systems. In: atp - Automatisierungstechnische Praxis, H. 11, S. 888–893.

Schlereth, Michael; Lauder, Marius; Rose, Sebastian; Schürr, Andy (2010): Concurrent Model Driven Automation Engineering. Building Engineering Tool Integration Systems. In: Automation 2010. Der Automatisierungskongress in Deutschland ; Kongress Baden-Baden, 15. und 16. Juni 2010 ; 11. Branchentreff der Mess- und Automatisierungstechnik. Düsseldorf: VDI-Verl. (VDI-Berichte, 2092), S. 7–10.

Anjorin, Anthony; Lauder, Marius; Schlereth, Michael; Schürr, Andy (2010): Support for Bidirectional Model-to-Text Transformations. In: Cabot, Jordi (Hg.): Workshop on OCL and Textual Modelling. Part of the ACM/IEEE 13th Intenational Conference on Model Driven Engineering, Languages and Systems (MoDELS'10). Oslo, Norway, 3rd October 2010 .

Lauder, Marius; Schlereth, Michael; Rose, Sebastian; Schürr, Andy (2010): Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. In: Bulletin of the Polish Academy of Sciences, Jg. 58, H. 3.

Rose, Sebastian; Lauder, Marius; Schlereth, Michael; Schürr, Andy (2010): A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In: Osis, Janis; Asnina, Erika (Hg.): Model-driven domain analysis and software development. Architectures and functions. Hershey, PA: Information Science Reference, S. 90–113.

Schlereth, Michael; Rose, Sebastian; Schürr, Andy (2009): Model Driven Automation Engineering – Characteristics and Challenges. In: Giese, Holger; Huhn, Michaela; Nickel, Ulrich; Schätz, Bernhard (Hg.): Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme V. 5. Dagstuhl-Workshop MBEES 2009 (22.04.2009 - 24.04.2009) (Informatik-Bericht, 2009-01), S. 11–25.

Schenk, Bernhard; Schlereth, Michael (2008): Modellgetriebene Entwicklung in der Automatisierungstechnik. In: Automation 2008. Lösungen für die Zukunft. VDI Berichte 2032. Düssdeldorf: VDI-Verl., S. 195–199.

Erklärung laut §9 PromO

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Datum Unterschrift