TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Elektrotechnik und Informationstechnik

Institut für Automatisierungstechnik

Entwurf nachhaltiger Lösungen zur Prozessvisualisierung in der industriellen Automatisierungstechnik mittels modellgetriebener Softwareentwicklung

Design of Sustainable Solutions for Process Visualization in Industrial Automation with Model-Driven Software Development

Stefan Hennig

Der Fakultät Elektrotechnik und Informationstechnik der Technischen Universität Dresden

zur Erlangung des akademischen Grades eines

Doktoringenieurs

(Dr.-Ing.)

genehmigte Dissertation

—— ❊ ——

Vorsitzender:	Prof. Dr.-Ing. Ralf Lehnert
Gutachter:	Prof. Dr. techn. Klaus Janschek Prof. Dr. Kris Luyten
Tag der Einreichung:	02.03.2012
Tag der Verteidigung:	07.06.2012

Acknowledgments

Successful research has at least three preconditions: (1) a reliable and carefree financial situation, (2) passionate scientific advice, and (3) many enthusiastic supporters. To meet the first requirement, this work was funded as "Landesinnovationspromotion" by the *European Social Fund* and the *Free State of Saxony*. This funding gave me the independence to research freely from conflicts of interests.

Prof. Dr. techn. Klaus Janschek, Managing Director of the Institute of Automation, Technische Universität Dresden, ensured the second requirement for successful research. I would like to thank him for his guidance and for the resources which made this work possible. With a balanced relationship between *encouragement* and *challenge*, he left me all the creative freedom while he ensured a successful conclusion.

Prof. Dr. Kris Luyten, Professor at the Expertise Centre for Digital Media, Universiteit Hasselt, Belgium, invited me to a research stay in his team. During this short time, we implemented several exciting ideas and I was able to learn a lot about the real business of research. Particularly, Kris' passion for science and for motivating support inspired me.

This work was realized as part of the research of the *Teleautomation* working group. It is based on collaborative work and a great number of shared ideas. Many thanks go to the fabulous team forming that group, particularly to the team leader *PD Dr.-Ing. Annerose Braune*. She asked countless critical questions and always had an open mind about unconventional approaches. I would like to thank *Evelina Koycheva*, *Matthias Freund*, and *Henning Hager* for the strong company and all the fruitful discussions. Henning had accompanied my project over several years at the narrowest. No one else has had such deep insight into my work.

The third requirement was met, on one side, by many students. Some parts of this work were greatly influenced by them, other parts were collaboratively developed. In particular, many thanks are devoted to *Nicolas Dingeldey*, *Alexander Witkowski*, *Ying Su*, and *Martin Halfter*. On the other hand, I am very grateful to the secretary of the institute, *Petra Möge*, and to the institute's backbone, *Matthias Werner*, for all the things they arranged for me during the past six years. Furthermore, I want to thank all the other doctorate students of the institute for having such a great time. Particular thanks go to *Thomas Kaden* who helped me

a lot with preparing the doctoral viva.

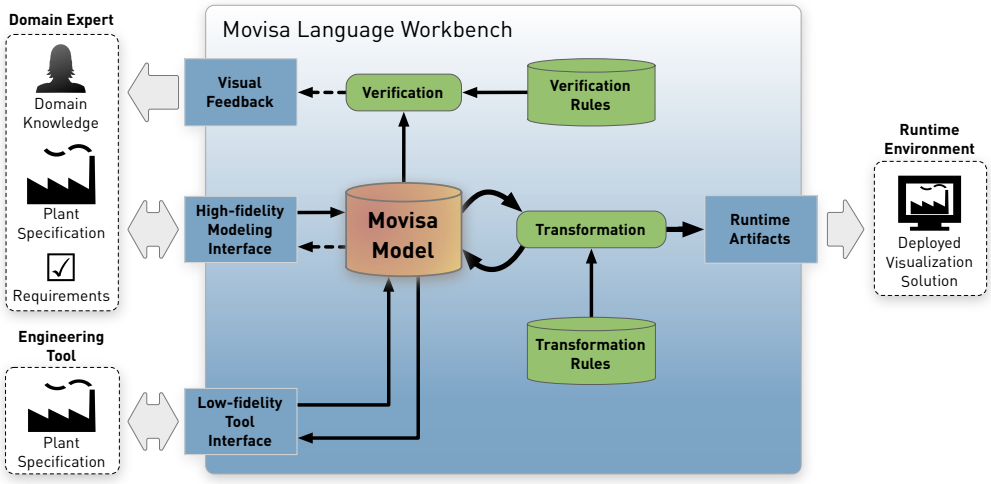
More important than getting technical support, however, is a strong support in "real life". Hence, I would like to thank my friends for always supporting me during all the years of hard work. Special thanks go to *Maik Biebl* for being so patient with me and for his insatiable interest in my work despite not understanding a single word. Special thanks also go to *Bärbel Wöhlke* for polishing my English.

"What you have started, must also be brought to an end!" As I grew up under this premise, my *parents*, Lorita and Günter, and my *brother* Sebastian have always given me the courage to finish what I had started and the freedom to make my own decisions. I am in great debt to them.

And finally, I thank *Katrin Holinski* for reminding me so often that there is a life beyond technology.

Abstract

Industrial facilities are supervised using dedicated *Supervisory Control and Data Acquisition* (SCADA) applications. These applications, however, suffer from being developed using platform specific terminologies which cause that their operative characteristics are strongly merged with aspects of the technical realization. Platforms executing these applications are characterized by short innovation cycles, thus, decreasing the life time of SCADA applications. Industrial facilities, however, are required to be in operation for decades which possibly requires repeated redevelopment of these applications even if the operative characteristics remain the same. *Model driven* techniques are promising design approaches to foster sustainability of SCADA applications: They separate operative characteristics from their technical realization using *Domain Specific Languages*.



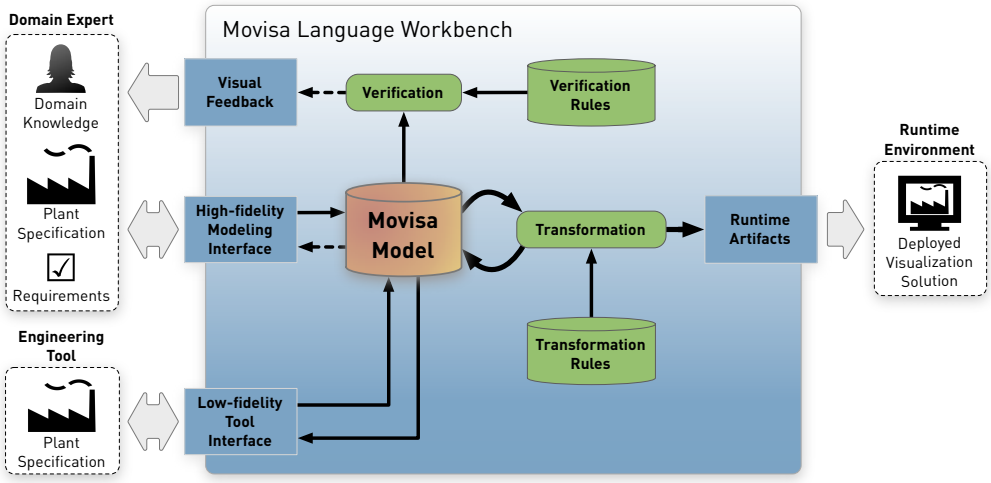
This thesis proposes the domain specific modeling workbench Movisa. Its core consists of a domain specific modeling language enabling to capture operative characteristics of SCADA applications. For this purpose, it contains building blocks to create user interfaces, process data and communication relationships with automation specific data servers, and it allows to express custom functionality through an Executable UML realization. *Language Constraints* and model-integrity checks allow to identify errors in early design stages and ensure the correctness of models. Transformation rules capture aspects of the technical realization: They allow to process Movisa models either to modify these models or to automatically create runtime artifacts. In this context, different kinds of transformations are

provided in order to support modelers in their assignments and, thus, to reduce the overall development effort. This complexity is encapsulated behind a high-fidelity modeling interface to be exploited by domain experts. It allows to solve problems with a common terminology that is very close to the respective solution space. Furthermore, engineering tools are able to populate MOVISA models via the low-fidelity tool interface.

Case studies from different fields of the domain *production automation* prove the language to be able to describe SCADA applications, thus, meeting related requirements of industrial automation. Sustainability of these applications can be ensured, among others, through automatic transformations, by reusing models and transformations in future projects and through having only one tool to master. The quintessence of this thesis is that even though model driven approaches are challenging with respect to provide effective tool environments, they are very promising means for creating sustainable software designs.

Kurzfassung

Unter *Supervisory Control and Data Acquisition* (SCADA) wird das Überwachen und Bedienen technischer Produktionsprozesse verstanden. Handelsübliche SCADA-Systeme erzeugen Lösungen auf Basis plattformspezifischer Terminologien. Daraus folgt eine enge Verzahnung funktionaler Inhalte mit Aspekten ihrer technischen Realisierung. Plattformen, auf denen SCADA-Lösungen genutzt werden, entstammen zunehmend dem Endverbrauchermarkt und unterliegen damit einer hohen Innovationsrate. Sich ändernde Plattformeigenschaften ziehen selbst bei gleichbleibenden funktionalen Inhalten eine Neuentwicklung dieser Lösungen nach sich. Die Einsatzzeit der SCADA-Lösungen ist somit verglichen mit der der industriellen Anlagen gering. Die modellgetriebene Software-Entwicklung bietet einen vielversprechenden Ansatz zur Erzeugung nachhaltiger SCADA-Lösungen, indem sie funktionale Inhalte von Aspekten ihrer technischen Realisierung auf Basis *Domänenspezifischer Sprachen* erlaubt.



Diese Arbeit schlägt zur Lösung der genannten Problemstellung die domänenspezifische Werkzeugkette MOVISA vor. Zentraler Bestandteil ist eine domänenspezifische Modellierungssprache, die funktionale Inhalte von SCADA-Lösungen zu beschreiben vermag. Dazu stellt sie Sprachmittel für die Beschreibung der Benutzungsschnittstelle sowie der Prozessdaten und Kommunikationsbeziehungen zur Verfügung. Aufgrund der Vielfalt technischer Prozesse enthält sie außerdem eine *Executable UML*-Realisierung, um die damit verbundenen Anforderungen zu adressieren. Mittel der Modellverifikation und -integritätsprüfungen ermöglichen die

Identifizierung von Fehlern bereits in frühen Entwurfsphasen und garantieren die Korrektheit der Modelle. Transformationsregeln enthalten die Aspekte der technischen Realisierung. Hinsichtlich einer automatischen Erzeugung der Laufzeitartefakte werden diese den MOVISA-Modellen zugeführt. Weitere Transformationen verarbeiten MOVISA-Modelle, um den Modellierer in seinen Aufgaben zu unterstützen und damit den Entwicklungsaufwand zu reduzieren. Die mit diesen Komponenten verbundene Komplexität bleibt dem Modellierer durch einen *high-fidelity* Arbeitsraum im vorgeschlagenen Werkzeug verborgen. Dies ermöglicht das Arbeiten mit einer Terminologie, die sich nah am Lösungsraum des jeweiligen Problems befindet. Über eine *low-fidelity* Schnittstelle erhalten Engineering-Werkzeuge Zugriff auf das MOVISA-Modell.

Fallstudien aus verschiedenen Anwendungsfeldern der Domäne *Produktionsautomatisierung* belegen, dass die vorgeschlagene domänenspezifische Sprache imstande ist, SCADA-Lösungen zu beschreiben. Die Nachhaltigkeit dieser Lösungen ist unter anderem durch automatische Transformationen, durch Wiederverwendung der Modelle und Transformationen in späteren Projekten sowie durch die Pflege nur eines Werkzeugs sichergestellt. Als Quintessenz dieser Arbeit wird festgestellt, dass modellgetriebene Ansätze zur Softwareentwicklung zwar vor dem Hintergrund der Bereitstellung effizienter Werkzeuge herausfordernd sind. Doch zeigen sie sich vielversprechend für den Entwurf nachhaltiger Softwarelösungen.

Contents

List of Figures

List of Tables

List of Listings

Nomenclature

API	Application Programming Interface
ASL	Action Specification Language
ATL	ATLAS Transformation Language
AUI	Abstract User Interface
CAE	Computer Aided Engineering
CAEX	Computer Aided Engineering eXchange
CRF	Cameleon Reference Framework
CUI	Concrete User Interface
DES	Discrete Event System
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
EGL	Epsilon Generation Language
EOL	Epsilon Object Language
EVL	Epsilon Validation Language
FCML	Facility Control Markup Language
FDA	U.S. Food and Drug Administration
FLEPR	Flexible Workflow for early User Interface Prototypes
FUI	Final User Interface
FUML	Semantics of a Foundational Subset for Executable UML Models
GUI	Graphical User Interface
HCI	Human Computer Interaction
HMI	Human Machine Interface
HTML	Hypertext Transfer Markup Language
JET	Java Emitter Templates
M2M	Model to Model (Transformation)
M2T	Model to Text (Transformation)
MARIA	Model-based lAnguage foR Interactive Applications
MBUID	Model Based User Interface Development
MDA	Model Driven Architecture
MDSD	Model Driven Software Development
MOF	Meta Object Facility
OAL	Object Action Language

OCL	Object Constraint Language
OMG	Object Management Group
OPC	OLE for Process Control *(Nowadays, OPC is used without referring to an abbreviation, as the importance of the OLE interface decreases.)*
OPC UA	OPC Unified Architecture
PLC	Programmable Logic Controller
PUC	Personal Universal Controller
QVT	Query/Views/Transformations
SCADA	Supervisory Control and Data Acquisition
Scrall	Starr's Concise Relational Action Language
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SVG	Scalable Vector Graphics
T&C	Tasks and Concepts
UCD	User Centered Design
UI	User Interface
UIML	User Interface Markup Language
UML	Unified Modeling Language
UsiXML	User Interface Markup Language
VBA	Visual Basic for Applications
WSDL	Web Services Description Language
XIML	eXtensible Interface Markup Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSD	XML Schema Definition
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformation
XUL	XML User Interface Language
XVCML	eXtensible Visualization Components Markup Language

Chapter 1

Introduction

Automation systems consist of technical processes and the required automation equipment [Lunze, 2008]. A technical process is, according to Johannsen (1993), a physical-technical or a chemical-technical procedure with material, energy, and/or information flows at input and output. Johannsen (1993) instanced a procedure on a milling machine with a raw workpiece and electrical energy as input, the processed workpiece, chips, and thermal energy as output. The automation equipment is composed of devices required to transform material and energy. These devices impact on the technical process, e.g. a servo motor moving the milling head.

Human operators are responsible for a safe operation of an automation system [Johannsen, 1993]. Hence, it is monitored and operated by human operators through appropriate *Human Machine Interfaces* (HMI), constituting the connection between the human operator and the automation system: It allows for monitoring the operative states of the automation system by presenting relevant information that have appropriately been prepared. Furthermore, it consists of input devices for entering information. In this way, human operators are enabled to have an impact on the technical process according to its actual state and given goals. Sheridan (1992) makes a distinction between *Human Computer Interaction* (HCI) and *Supervisory Control*: While in HCI one uses computers to operate other computers or databases as end objects, in *Supervisory Control* computers are only mediators between a technical process and human supervision. Sheridan (1992) defines *Supervisory Control* as follows: "[...] one or more human operators are intermittently programming and continually receiving information from a computer that itself closes an autonomous control loop through artificial effectors and sensors to the controlled process or task environment"[1]. Figure 1.1 shows the basic paradigm

[1] Cassandras and Lafortune (1999) introduce *Supervisory Control* in automata theory for a given *Discrete Event System* (DES) "whose behavior must be modified by feedback control in order to achieve a given set of specification." This DES is modeled by a graph G with an event set E. If the behavior of G is not satisfactory, it must be controlled by a supervisor S. S observes all events that G executes and then, "S tells G which events in the current active event of G

behind this definition. Appendix A exemplarily presents concrete supervisory control solutions.

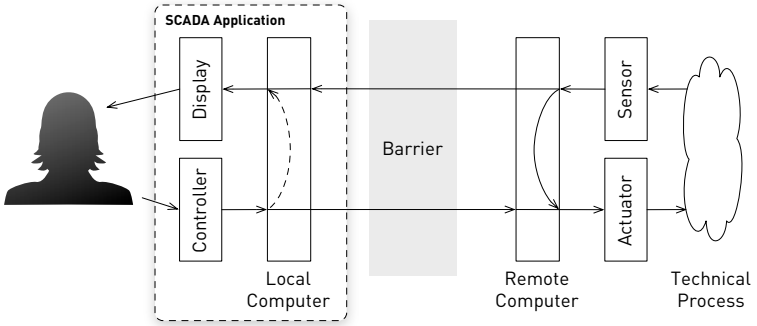


Figure 1.1: Basic Supervisory and Control Paradigm (based on [Sheridan, 1992]).

A remote computer closes a control loop by observing values actually measured by sensors and by setting particular actuators according to these values and given (control) programs. Human operators supervise this process through *graphical displays* representing the technical instrumentation and employ appropriate controllers to modify parameters in the remote programs. Both, the operator and the remote device, might be separated by "a barrier of distance, time, or inconvenience" [Sheridan, 1992]. To implement this general concept, *Programmable Logic Controllers* (PLC) that act independently and close to the process are usually used as remote computers. *Supervisory Control and Data Acquisition* (SCADA) applications constitute the local part depicted in Figure 1.1. Bailey and Wright (2003) refer SCADA "to the combination of telemetry and data acquisition": It encompasses the collection of relevant information, carrying out any necessary analysis and preparing that information to be presented on a number of operator screens or displays. Required control actions are then conveyed back to the process. A SCADA application is, according to Daneels and Salter (1999), a purely software package that is positioned on top of hardware. Thus, SCADA applications form the *Human Machine Interface* for process visualization.

Definition 1.1: *A* Visualization System *is a software tool to develop SCADA applications. A* Visualization Solution *is a particular SCADA application tailored to supervise a concrete technical process, thus being the* User Interface *to this process.*

are allowed next."

In the following, Section 1.1 explains the necessity for a new approach to the development of visualization solutions and introduces the solution proposed in this thesis. Section 1.2 provides an overview of structure of this thesis.

1.1 Problem Statement and Aims

Vital requirements for graphical human machine interfaces for process control are among others *efficiency*, *ergonomics*, and that they "shall be so designed as to allow the operator to perform [her] activities in accordance with [her] capabilities, skills and needs, as required to achieve [her] objectives" [VDI/VDE, 2005]. For this purpose, the guideline VDI/VDE (2005) recommends to involve operators into the design phase. Therewith it proposes a *User Centered Design* (UCD) approach, thus entailing a significant amount of the overall system design and development effort. On the other hand, platforms executing SCADA applications in industrial automation are characterized by diversity as illustrated in Figure 1.2.

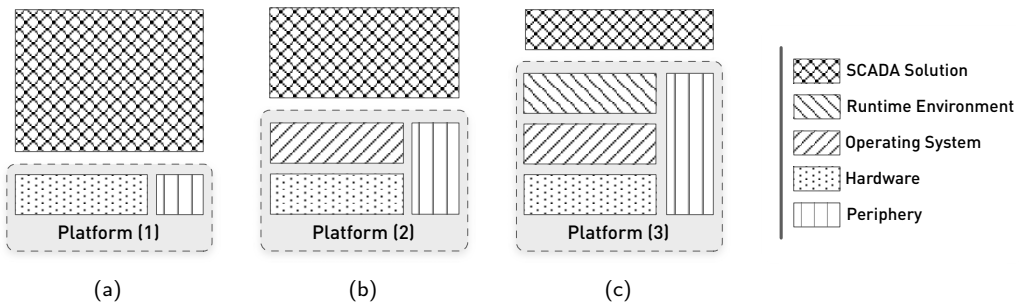


Figure 1.2: Illustration of three SCADA applications operated on different platform configurations: (a) shows an embedded device with a native SCADA realization to be exploited close to the process, (b) presents a mobile version for remote access, and (c) illustrates a classical installation of a SCADA application to be operated on a workstation in control rooms. These platform configurations can also be combined.

Currently, visualization solutions are developed using platform specific tools and terminologies, such as a specific programming language. As a consequence, operative characteristics are strongly merged with aspects of their technical realization. Thus, a human machine interface has to be redeveloped for and tested on each platform using different and probably incompatible tools, even if the operative characteristics remain the same. Moreover, as Menzel et al. (2003) state, the situation is exacerbated by the fact that standard end-user platforms are used

almost exclusively. These platforms are characterized by short innovation cycles decreasing the life time of SCADA components to less than five years due to regular releases of new software versions [Menzel et al., 2003]. Given the fact that industrial facilities are required to be in operation for decades and given the complexity of SCADA applications in order to ensure correctness, reliability, and usability, new design approaches are demanded to suit the rapid development of the platforms.

Model driven techniques are promising design approaches, as they express operative characteristics through a platform independent terminology that is based on models. Technical aspects are separated into transformations as carriers of platform specific terminologies. For the given situation, model driven approaches enable to capture the operative characteristics of a SCADA solution by creating a platform independent model. Deploying this solution to a particular platform requires an appropriate transformation that translates the platform independent terminology, provided by the model, into a platform specific terminology. If a compatible platform evolves over time, e.g. through an update of the operating system, causing incompatibility or to equip another or a new platform with this existing SCADA solution, only suitable transformation is required while the respective model remains unchanged. Consequently, operational characteristics need to be expressed and tested for correctness, reliability, and usability only once. Tested and correct transformations are expected to produce always correct platform specific runtime solutions. Additionally, model driven approaches offer the following more general benefits:

(1) Functional aspects can be reused in future projects, even if they aim at another platform.
(2) Technical realizations can be reused in future projects, even if the functionality is a different one.
(3) High-quality and reproducible solutions can be generated through tested transformations.
(4) Various technical solutions can automatically be created from a single functional description, simply by invoking another transformation.
(5) The repertoire of required tools can be slimmed down.

Nichols, Chau, and Myers (2007) have already proven the viability of model driven approaches for the development of user interfaces for office applications. Aquino et al. (2010) additionally proved that model driven development procedures are promising even for automatically deploying user interfaces to different devices. Visualization solutions in industrial environments, however, are characterized by specific requirements and constraints that the aforementioned approaches to model office solutions do not meet, mainly due to the fact that office applications are

connected to databases, whereas industrial solutions are connected to technical processes.

> This thesis transfers the aforementioned advantages of model-driven approaches into the domain of industrial *production automation*[2]. It proposes a *Domain Specific Language* which enables the development of sustainable visualization solutions. Other than model-driven approaches for office applications, this modeling languages is tailored to meet the requirements of industrial automation: (1) It defines a sufficient set user interface components with appropriate *representation*, *animation*, and *interaction* properties. (2) It provides a solid abstraction to the variety of industrial automation specific process communication means. (3) It contains an *Executable UML* realization to take into account the diversity of industrial processes and their individual requirements.

This thesis contributes the domain specific modeling workbench MOVISA, as depicted in Figure 1.3: MOVISA models capture operative characteristics of visualization solutions. Transformation rules capture aspects of the technical realization. The verification tool ensures the correctness of models and the transformation tool processes MOVISA models either to modify these models or to automatically create runtime artifacts. This complexity is encapsulated behind a high-fidelity modeling interface to be exploited by domain experts. Engineering tools are able to populate MOVISA models through low-fidelity tool interface.

[2] Johannsen (1993) brings the fields of *process industries*, *factory automation*, and *energy supply systems* under the umbrella of production automation together.

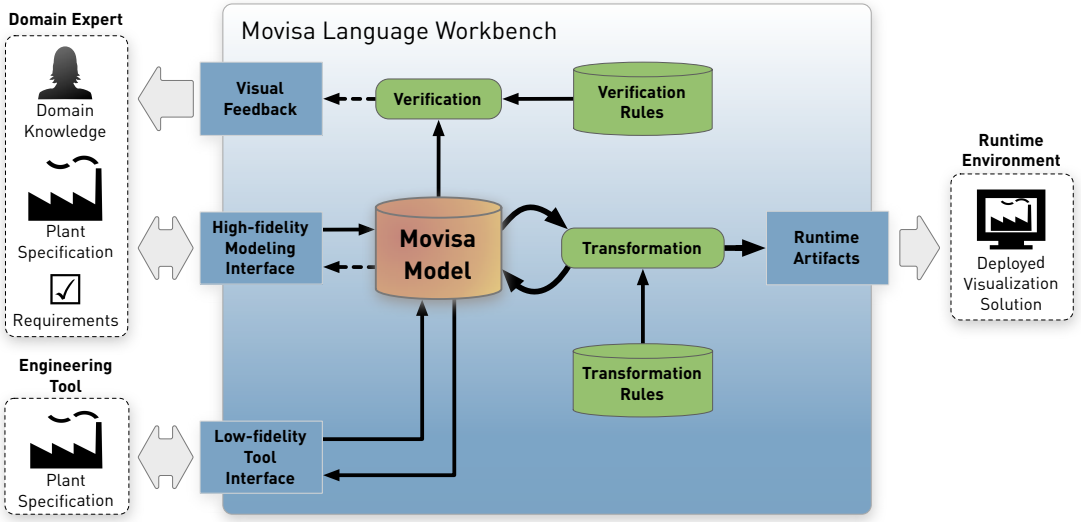


Figure 1.3: Basic concept, architecture, and functionality of the proposed MOVISA modeling workbench.

1.2 Thesis Structure

This thesis is structured as follows:

Chapter 2 gives an overview of the state of the art and its particular background. This chapter concludes with a detailed requirements definition used for pointing out deficiencies of the state of the art approaches and how this thesis contributes to them.

Chapter 3 formalizes the *Language Model* of the *Domain Specific Language* MOVISA by abstracting its *Target Domain* into the *Core Language Model*. Section 3.3 explains *Language Constraints* and Section 3.4 presents the *Language Behavior* definition.

Chapter 4 works out and discusses a concrete syntax notation that enables modelers to work and to think their domain.

Chapter 5 discusses required aspects of a modeling workbench for creating, using, and maintaining models of the language that was created in Chapter 3 which encapsulates the complexity of the model driven approach.

Chapter 6 evaluates the feasibility of the MOVISA modeling workbench by exploiting it on representative case studies.

Chapter 7 elaborates a transformation based framework enabling to incorporate the MOVISA modeling workbench in higher-level engineering procedures.

Chapter 8 draws the conclusions of the findings of the previous chapters.

Chapter 2

State of the Art

Providing an approach to the development of *Visualization Solutions* for supervising technical processes in a way that allows the reusing of the engineered solutions on other possibly incompatible platforms or environments only through transformation is influenced by the methods and techniques emerged from different research fields. This chapter introduces relevant contributions, discusses the state of the art and emphasizes on its deficiencies, from which concrete requirements for the proposed modeling workbench are deduced. First, Section 2.1 gives a brief overview of the engineering process of automation systems and emphasizes on the phase in which the development of visualization solutions takes place. Section 2.2 introduces the research field of *Model Based User Interface Development* that contributes to a systematic development of user interfaces employing models for expressing design knowledge. The research field *Model Driven Software Development*, introduced in Section 2.3, provides methodologies and techniques to generate virtually arbitrary artifacts from formal models. Its core concept *Domain Specific Languages* will be separately discussed in Section 2.4. Section 2.5 draws the conclusions, specifies requirements for the thesis work, and based on this, it presents the contributions of this thesis.

2.1 Human Machine Interface Engineering in Automation

Elementary characteristics of *SCADA applications* are (1) *Monitoring and Operating* technical processes by human operators through *Human Machine Interfaces*, (2) *Alarm Management* to immediately react on exceptional process states, and (3) *Archiving Process Data* for assessment [Zacher and Wolmering, 2009]. In the overall planning and development phase of automation facilities, these visualization solutions are build at the very end [Urbas and Doherr, 2011] (see Figure 2.1) due to many interdependencies of engineering data. For instance, visualization solutions gather process data from PLCs to which they are connected through an automation

specific field bus[1] in order to genuinely reflect the actual process states and to provide means for managing, logging, or archiving process data. Hence, this entails the definition of very concrete data models, but the required data specifications only are available in late engineering phases. Using dedicated SCADA development tools, these data specifications are converted into a format specific to the particular tool, being the basis for developing the visualization solution.

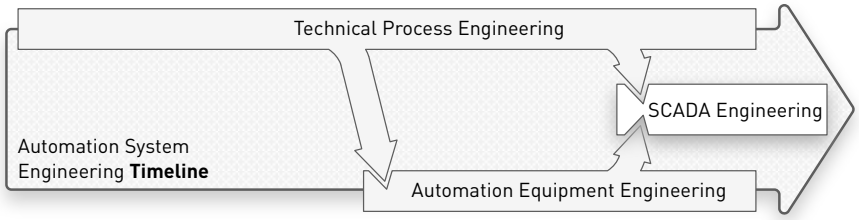


Figure 2.1: Schematic diagram of the engineering timeline of an *Automation System* (loosely based on [Urbas and Doherr, 2011]): SCADA engineering takes place at the very end.

Zacher and Wolmering (2009) emphasize that SCADA tools typically comprise of both an engineering and a runtime environment. Visualization solutions that are developed with a particular engineering environment require the matching runtime environment for execution. According to Daneels and Salter (1999), engineering environments support *Domain Experts* at least with the following features:

(1) A data model configuration tool that allows configuring the process data to be exchanged between the visualization solution and the PLC in order to be presented to an operator or to be archived.

(2) Graphics editor that provides standard and automation specific interaction elements. This editor allows for arranging user interface components on screen and to link them to process data.

(3) A scripting language editor enables to add custom and project specific functionality to a visualization solution. Due to the diversity and possibly uniqueness of industrial processes, it is not worthwhile to provide standard building blocks for each foreseeable requirement. Consequently, functionality besides the basically provided one can be realized using a programming language. For instance, routines for manipulating process data in a specific manner can be defined.

[1]Field busses enable a high-available and real-time capable information exchange.

Definition 2.1 (Domain Expert)**:** *A* Domain Expert *is an automation engineer with advanced knowledge about the particular technical process, physical backgrounds, possible events and hazardous situations as well as details on the product to be manufactured. Domain experts have sound programming skills with respect to using the aforementioned engineering environments. This implies being proficient in configuration and (visual) programming tasks as well as in scripting and programming languages.*

WinCC [Siemens, 2011], *InTouch* [Wonderware, 2011], *ProWin* [OHP, 2012], and *Genesis64* [ICONICS, 2011] are visualization systems characterized by comprehensive engineering environments and efficient runtime environments including the aforementioned features. Additionally, these systems offer interfaces for a web-based remote operation, also to broaden the range of possible target platforms. Other visualization systems allow to create pure web-based visualization solutions. Basically, two different kinds of products can be distinguished: (1) Vendors of PLCs equip their hardware with web servers and provide tools to create simple but dynamic websites for monitoring and modifying the parameters of the particular PLC. *DigiWeb* [Digitronic, 2011] can be seen as a representative. (2) More flexible and powerful web-based solutions can be provided, e.g. by *atvise webMI* [atvise, 2011] or by the framework *ACPLT/HMI* [Schmitz and Epple, 2007]. However, all these web-based approaches share that they define proprietary data exchange formats between the web browser and the vendor-specific web server.

Concluding, several powerful tools for monitoring and operating the operative states of technical processes were established in industrial automation. They support engineers with comprehensive engineering environments and contain vast libraries with predefined components. However, a resultant visualization solution always depends on a specific platform, either due to vendor-specific runtime environment or because of proprietary infrastructure components. Visualization solutions for different platforms need to be developed with different tools. This might not only lead to an inconsistent behavior, but also to an increase of the overall project time and cost. This thesis contributes to overcome these shortcomings.

2.2 Model Based User Interface Development

Redeveloping a user interface (and particularly a visualization solution) for each platform with the available terminology, e.g. in terms of a specific programming language, leads to unnecessary efforts and might cause inconsistent behavior across several platforms [Eisenstein, Vanderdonckt, and Puerta, 2001]. Therefore, model

based approaches to the development of user interfaces for different platforms were established and combined in the research area of *Model Based User Interface Development* (MBUID). According to Paternò (2000) and, more recently, according to Calleros et al. (2010), MBUID fosters the specification of user interfaces without making any implementation specific decisions. Implementation specific solutions will be derived from such specification manually or (semi-)automatically using specific tools. In that way, several platforms can be equipped with user interfaces that are based on the same user interface specification. Thus, MBUID forms a key pillar in this thesis to solve the problems identified in Section 1.1.

MBUID approaches use models at different levels of abstraction to specify the relevant aspects of user interfaces using models which can comprise, according to Benyon, Green, and Bental (1999), (1) functional models, (2) dynamic models, or (3) structural models: Nichols and Myers (2009) propose the *Personal Universal Controller* (PUC) to describe the functions and features of office appliances using an XML[2]-based language (functional model). A specific runtime environment (the PUC rendering engine) loads models of this language and automatically creates a user interface from it. PUC can be categorized as an abstract approach. Abrams et al. (1999) present the *User Interface Markup Language* (UIML) that is an XML-based language to express the functional specification of user interfaces as a "vendor-neutral, platform independent, [and] canonical representation" [Helms et al., 2009]. It provides a set of models to define the structure, the presentation (both structural models), and the behavior (dynamic model) of a user interface. A vocabulary maps an individual UIML model to a specific platform dependent toolkit. UIML models are concrete models, as they describe concrete interaction modalities and agree on a certain structure of the user interface. The *eXtensible Interface Markup Language* (XIML) [Puerta and Eisenstein, 2001] as well as *Mobi-D* [Puerta, 1997] combine an equal set of models, namely: a *presentation model* (structural), a *dialog model* (dynamic), a *task model* (functional), and a *user model* to a single *interface model*. Thus, these approaches use models that span several levels of abstraction from abstract ones (task model) to concrete ones (presentation model).

Definition 2.2: *The more abstract a user interface model is, the more independence from certain platforms can be gained. Functional and dynamical models are completely platform independent thus being* abstract models. *Structural models depend on specific interaction modalities and are tangible enough to get an impression of the final look thus being* concrete models.

Because each of the aforementioned user interface modeling techniques define

[2]eXtensible Markup Language

models that underly different levels of abstraction, the Cameleon Reference Framework (CRF) was introduced by Calvary et al. (2003) to support a classification of these models as well as the methods and processes to employ them. The CRF is a conceptual generic framework fostering systematic development of user interfaces, as it provides a "common vocabulary within the HCI community to discuss and express different perspectives on a user interface" [Coutaz, 2010]. Figure 2.2 depicts a simplified version of this framework as introduced by Limbourg et al. (2005).

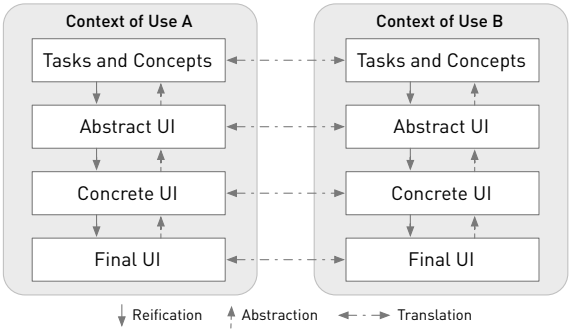


Figure 2.2: A simplified version of the Cameleon Reference Framework [Calvary et al., 2003], as proposed by Limbourg et al. (2005).

The CRF introduces the *Context of Use* that defines a structured information space [Calleros et al., 2010] in terms of user, platform, and environment: (1) The user interacts with the particular system through the user interface. (2) The platform is a set of resources on which the user interface will be executed. (3) The environment comes along with conditions that might influence the interaction capabilities of a system[3]. As Figure 2.2 illustrates, a user interface of a particular context of use is structured by models classified with the following levels of abstraction:

(1) *Tasks and Concepts* (T&C) defines functional models that describe the user's task to be accomplished through the user interface and necessary domain objects to be manipulated by these tasks. The domain objects form the actual data to be presented or modified.

(2) *Abstract UI* (AUI) embraces dynamical models that describe a set of abstract interaction objects independently from any concrete interaction modality [Vanderdonckt, 2005].

[3]For instance, a strong solar irradiation can reduce the readability of computer screens.

(3) *Concrete UI* (CUI) specifies functional models of the UI at a level that is independent from a specific programming language or toolkit. Yet, they depend on specific platforms in terms of input and output modalities.
(4) *Final UI* (FUI) represents the executable user interface that depends on a platform in terms of software stacks. For instance, SCADA systems such as WinCC (see Section 2.1) produce visualization solutions that reside at this level of abstraction.

Transformations form the transition between the models of the CRF. Calvary et al. (2003) distinguish two types of transformations:

Definition 2.3: Vertical transformations *are used to gain more concrete models from abstract ones (reification) and vice versa (abstraction).* Horizontal transformations *are used to move a model to another context of use (translation) within the same level of abstraction.*

For instance, to deploy a visualization solution to a platform containing a different software stack, a vertical transformation from CUI models to the FUI code is required. If the platform is characterized by a different screen size, the CUI model needs to be transformed beforehand to another context of use by a horizontal transformation. Therefore, the definition of the term "platform" needs a differentiated consideration: Calvary et al. (2003) characterize platforms by properties that determine a particular context of use, e.g. their input and output interaction means and other hardware related resources, such as screen size. Wagelaar and Jonckers (2005), however, explicitly distinguish platforms also in terms of their containing software stacks. Those properties do not establish another context of use.

Definition 2.4: *The term* Platform *is characterized in this thesis by the terms* hardware platform *and* computing platform. *A hardware platform establishes its individual context of use. A computing platform can be seen as a specific* Profile *which is characterized by particular software components. Such profile can be shared among hardware platforms thus even thought a new context of use has been established, vertical transformations can be reused.*

The *User Interface Markup Language* (UsiXML) [Limbourg et al., 2005] is a sound reference implementation of the CRF. It provides an XML-based modeling language for each layer of abstraction and a transformation system that is based on graph transformations. *MARIA XML* [Paternò, Santoro, and Spano, 2009] also covers all levels of the CRF. Additionally, it introduces an extensible data model as well as an event model. More industry driven approaches are the *XML User*

Interface Language (XUL) [Feldt, 2007; Goodger et al., 2011] or the *eXtensible Application Markup Language* (XAML) [Microsoft, 2011]. Both approaches belong to the so called *platform-integrated* techniques: they describe structural parts of the user interface using an XML dialect. The dynamic part is formulated using a particular programming language. Therefore, they actually reside at the FUI level of the CRF.

To sum up, MBUID proposes to use models to support developers in managing the complexity of interactive applications [Paternò, 2000] by providing different points of view on those applications [Benyon, Green, and Bental, 1999]. Using the Cameleon Reference Framework, existing user interface models can be uniformly classified. Furthermore, it also does provide a common terminology among developers and designers as a basis to discuss new concepts — particularly the approach presented in this thesis — and to share knowledge across disciplines.

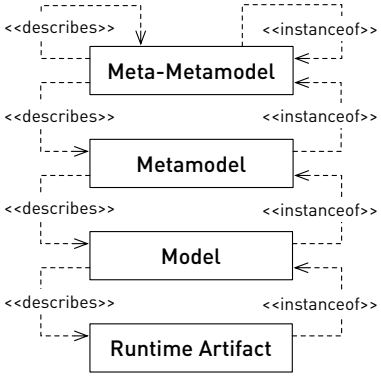
2.3 Model Driven Software Development

The *Model Driven Software Development* (MDSD) procedure is an open and integrative approach that combines, according to Stahl et al. (2007), techniques to automatically generate arbitrary artifacts. Using a common terminology, models enable the specification of software systems at a functional level without the need for decisions about technical details. In this way,the associated problem can be addressed in a pure functional solution space. Some transformations add technical details, such as platform specifics, to models that as a consequence, "can be deployed in various software environments without change" [Mellor and Balcer, 2002]. Hence, *operative characteristics* of a software system are clearly separated from their technical realization [Zeppenfeld and Wolters, 2006] — with this, the MDSD forms another key pillar for ensuring the sustainability of visualization solution designs.

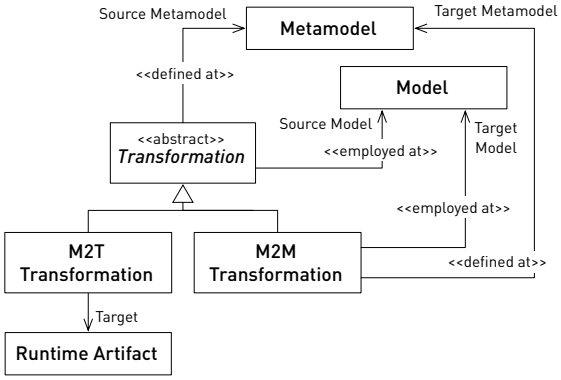
Definition 2.5 (Variable vs. fixed operative characteristics)**:** *Aspects that characterize an individual system are referred to as* Variable Operative Characteristics *and are stored in models.* Fixed Operative Characteristics *are shared among all systems to be produced by a particular transformation and are captured by it.*

A vital requirement for automatic transformations is the use of *formal* models. They allow the description of the relevant domain completely, precisely, and without contradictions. Moreover, a formal model is required to be compliant to a formal metamodel, whereas "[a] metamodel is simply a model of a modelling language" [Mellor et al., 2004]. It defines, according to Stahl et al. (2007), available modeling elements, their relation to each other as well as constraints and

modeling rules. The required elements of the metamodel are defined in its own metamodel — the meta-metamodel. Theoretically, any number of these meta levels can be introduced. Practically, meta-metamodels are expressed in a self-descriptive manner. Figure 2.3a illustrates the resulting meta levels schematically. Similarly, transformation languages are required to be formal languages based on a meta-metamodel to ensure syntactical correctness of transformations and to facilitate verification of the generated results.



(a) Models at different levels of abstraction, as introduced by Petrasch and Meimberg (2006).



(b) Terminology of transformations according to Stahl et al. (2007).

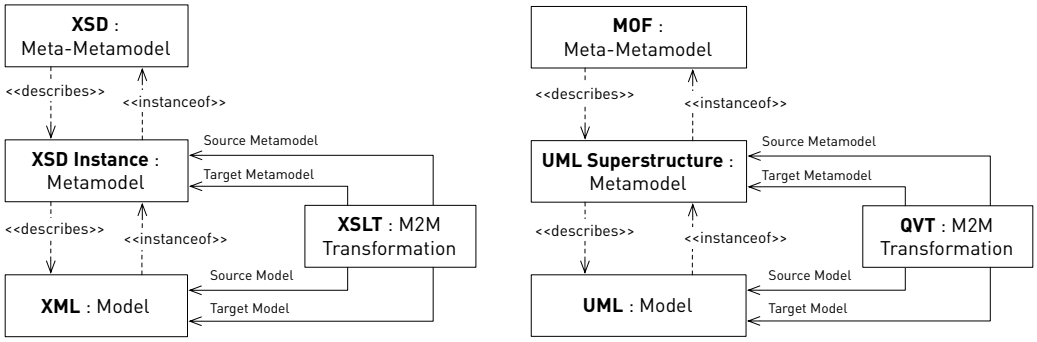
Figure 2.3: Illustration of the core concepts of MDSD: (a) shows the commonly used relation between models; (b) connects these models through transformations.

Definition 2.6: *Transformations form a link between models and other artifacts that can be either other models in case of* Model-to-Model (M2M) Transformations *or platform specific runtime artifacts in case of* Model-to-Text (M2T) Transformations *[Stahl et al., 2007].*

Transformations are defined at the metamodel level by addressing the elements to be transformed. They are employed at the model level (see Figure 2.3b). To define transformations, Mens, Czarnecki, and Gorp (2006) distinguish between *declarative* and *operational* (imperative) transformation mechanisms. While declarative approaches have their strengths in compactness and maintainability because of hiding procedural details, operational approaches may unfold their advantages when incremental model updates are required [Mens, Czarnecki, and Gorp, 2006].

As the aforementioned concepts are independent of any concrete modeling

technique — formal models can be defined among others using XML or UML[4] —, the OMG[5] provides an industry specification with the *Model Driven Architecture* (MDA, [Mellor et al., 2004; Miller and Mukerji, 2003]) to unify model-driven processes and tools. Basically, it introduces the *Meta Object Facility* (MOF, [OMG, 2010]) as meta-metamodel and proposes UML as modeling language as well as QVT[6] as transformation technique. Figure 2.4 compares two different modeling techniques using the terminology introduced in Figure 2.3.



(a) XML as one MDSD compliant modeling technique.

(b) Modeling techniques defined by the MDA.

Figure 2.4: Comparison of two MDSD compliant modeling techniques: (a) employs the XML technology stack as formal modeling technique; (b) employs the techniques defined in the MDA specification by the OMG.

Concluding, the MDSD constitutes a fundamental basis for the developments of this thesis by defining general concepts like the usage of metamodels and transformations. Pietrek et al. (2007, p. 14), however, criticize the imprecise MDA specification to be only partially suitable in practice. With the *Eclipse Modeling Framework* (EMF) [Eclipse, 2011c; Steinberg et al., 2008], however, a more pragmatic interpretation of the OMG standard is available. EMF uses *Ecore* as meta-metamodel. Since it can be seen as quasi-standard, a myriad of plugins was build around EMF. For instance, the *Epsilon* [Epsilon, 2011] model management suite most recently gained significance. It provides, among others, a validation language and generation languages for M2T (EGL, [Rose et al., 2008]) as well as M2M transformations. Nevertheless, the MDSD and its standardized interpretation

[4]Unified Modeling Language, [OMG, 2010b]
[5]Object Management Group, [OMG, 2010a]
[6]Query/Views/Transformations [OMG, 2011a]

MDA are still in an "early adoption phase" [Mohagheghi et al., 2009], as several challenges need still to be solved. For instance, Teurich-Wagner (2004) states that executable source code often cannot be generated completely, and thus, require manual post-treatment. The reason for this is the incapability of modeling languages to describe the relevant domain precisely enough and without contradictions.

2.4 Domain Specific Languages

Domain Specific Languages (DSL) are modeling languages tailored to a specific problem domain. Therefore, they are capable of describing a particular subject matter without contradictions and precisely enough to successfully employ the MDSD process (see Section 2.3). Thus, they form its core concept when it comes to automatically generating executable code so that models can be treated as *first-class artifacts*. Fowler (2011) distinguishes between external and internal DSLs: Internal languages enhance existing languages through domain specific concepts. External languages are stand-alone.

Definition 2.7: *(Domain) A* Domain *is, according to Stahl et al. (2007), a clearly defined realm of interest or knowledge.*

Like natural languages, a computer language consists of a *syntax* with its *semantics* and *presentations*. Strembeck and Zdun (2009) provide a formal definition of DSLs (see Figure 2.5): A *Language Model*[7] forms the abstraction of the DSL's target domain, representing the metamodel (comp. Figure 2.3). It is composed of (1) the *Core Language Model* that defines available modeling elements and relations between them; (2) *Language Model Constraints*[8] defining invariants on modeling elements; and (3) the *Language Behavior*[9] expressing the meaning of the DSL and particularly, how "the elements can interact at runtime" [Strembeck and Zdun, 2009]. One or more *Concrete Syntax* notations — the language's presentations — constitute the interface between the modeling language and the modeler, as they are used to express and to maintain models. Stahl et al. (2007) stress the importance of the design of concrete syntax notations in order to enable modelers to work and think in their particular domain. Fowler (2011) asks them to foster clarity for the modeler. According to Strembeck and Zdun (2009), it is a design goal of

[7] Also referred to as *abstract syntax* by some authors (e.g. [Kleppe, 2009]).
[8] Also referred to as *static semantics* [Demirezen, 2009; Stahl et al., 2007] or *structural semantics* [Chen et al., 2005].
[9] Also referred to as *dynamic semantics* [Demirezen, 2009; Ruscio et al., 2006] or *behavioral semantics* [Chen et al., 2005].

central importance to keep them simple and intuitive. To simplify the reading process, concrete syntax notations should, according to Ottensooser et al. (2012), provide a good cognitive fit to the thinking style with respect to the problem being solved: "Ideally, there should be no transformation between the notation-based representation to the mental representation" Ottensooser et al. (2012).

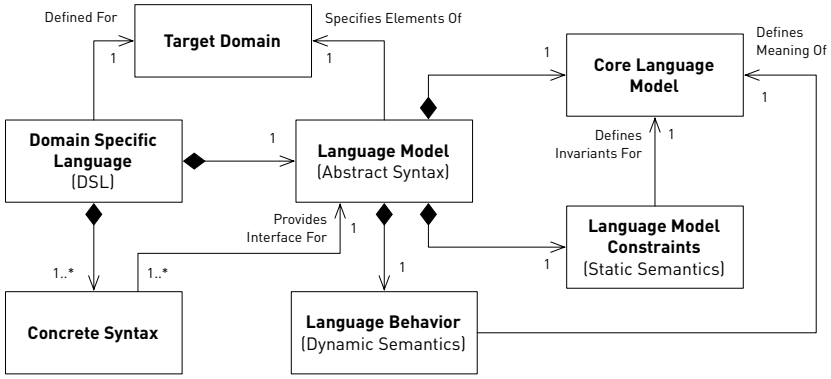


Figure 2.5: Artifacts of Domain Specific Languages as formalized by Strembeck and Zdun (2009).

As the MDSD requires formal models (see Section 2.3), the language model — the metamodel of the modeling language — needs in turn to be compliant to a metamodel. Therefore, choosing an appropriate meta-metamodel is crucial. Two important representatives are (1) the *XML Schema Definition* (XSD) [Thompson et al., 2011] for XML-based models and (2) the *Meta Object Facility* (MOF) [OMG, 2010] or Ecore [Steinberg et al., 2008], respectively, not only for UML-based models. The following discusses the aforementioned components of a DSL in more detail exemplarily using both MOF and Ecore as meta-metamodel.

Core Language Model

Basically, there are three different strategies to define the core language model using MOF/Ecore:

(1) Extending UML using its Profile Mechanism (lightweight extension of UML [Frankel, 2003]) to create a UML Profile: *stereotypes* and *tagged values* have to be defined that extend appropriate elements of the UML metamodel.

(2) The UML metamodel can be extended at MOF or Ecore level (heavyweight extension of UML [Frankel, 2003]), as the UML metamodel is defined via MOF or Ecore, respectively.

(3) Using MOF or Ecore, respectively, to create a core language model from scratch. This technique is particularly advantageous if the constructs of the intended modeling language "don't fit easily into any of the UML modeling paradigms" [Frankel, 2003].

Language Model Constraints

Not every invariant of the core language model can directly be expressed using means provided by the meta-metamodel. As an example, the presence of a particular property is ensured directly, e.g. by Ecore. Defining that the value of this property must be within a certain range is, however, not supported. Those invariants can be expressed with the *Object Constraint Language* (OCL) [OMG, 2011b]. As OCL is a side-effect-free constraint language, it does not allow to specify constraints of any complexity. Hence, the *EMF Validation Framework* [Eclipse, 2011a] enables the incorporation of constraints written in the *Java* programming language. Additionally, the *Epsilon Validation Language* (EVL) [Eclipse, 2011g] and *openArchitectureWare Check* [oAW, 2011] provide more convenient ways to formulate arbitrary constraints on Ecore-based metamodels.

Language Behavior

The language behavior defines the meaning of the DSL by explaining (1) the data to be processed, (2) the processes that handle this data, and (3) the relationship between both [Kleppe, 2009]. According to Kleppe (2009), the following techniques can be used to express the language behavior:

(1) Denotational semantics use mathematical statements.
(2) Pragmatic semantics demonstrate the behavior of the DSL by executing a reference implementation.
(3) Translational semantics translate the DSL into another language.
(4) Operational semantics can be seen as a series of snapshots of an abstract state machine.

Denotational semantics are the most complex way to describe semantics of a modeling language requiring a strong mathematical background. Executing a reference implementation is, according to Kleppe (2009), the simplest way to determine the meaning of a DSL. However, the more comprehensive a DSL is, the more difficult and error-prone it is to understand the entire behavior of a particular modeling language, because lots of conditions have to be considered. Kleppe (2009) states that translational and operational approaches are the most efficient ways to express the language behavior. Since the objective of MDSD is to create

runtime artifacts from models through transformations, the *translational semantics* is a convenient way to define the language behavior using transformations (see Section 2.3). They define the mapping between individual modeling elements and the respective parts of a particular runtime artifact.

Strembeck and Zdun (2009) summarize the aforementioned techniques as *detailed behavioral models*. Moreover, they propose less formal methods using *high-level control flow models*, such as *UML Activities* or precise textual specifications.

Concrete Syntax

Concrete syntax notations can be classified into graphical and textual representations. Both representations have strong advantages. Textual representations can be readily understood, which "can lead to significant reading effort for non-experts" [Ottensooser et al., 2012]. They are fast and simple to create and to maintain — best with a primitive text editor. Collaboration of teams often requires comparison of two or more different model versions. Comparing plain text is usually easy. The *Human Usable Text Notation* (HUTN) [OMG, 2004] allows to define a textual concrete syntax for MOF-based models. *EMFText* [EMFText, 2011] provides textual concrete syntax notations for Ecore based models based on the HUTN specification.

In contrast, even though graphical representations "require readers to have learnt the semantics of its symbols", they have strengths in terms of their efficient information processing [Ottensooser et al., 2012]. To visualize hierarchies and structures, two-dimensional — graphical — representations are more suitable: Modelers are able to capture data-flows, parallelism, and other complex relationships faster and more reliable in graphically noted models. Furthermore, they are the better basis for team discussions. The *Graphical Modeling Framework* (GMF) [Eclipse, 2011h] provides tools to equip Ecore based models with a graphical syntax. Graphical syntax notations for MOF based models can, for instance, be created using *MagicDraw* [No Magic, 2011]. It adds an additional layer to a UML Profile which forms the graphical concrete syntax [Silingas et al., 2009]. However, this graphical syntax is limited to small icons to be assigned to existing UML symbols. The tools *Graphical Modeling Environment* (GME) [GME, 2011] and *MetaEdit+* [MetaCase, 2011] form complete *Integrated Development Environments* for graphical DSLs. Both tools introduce a proprietary meta-metamodel. Exchanging models created with one of these tools between other tools is not possible. This strongly limits the options for choosing appropriate transformation engines.

Discussion

Obviously, it poses efforts to always create a modeling language only for limited problem domains: Developers need both deep knowledge about the problem domain and language definition expertise [Mernik, Heering, and Sloane, 2005]. Modelers always need to learn new languages, even though they are far more simple than general purpose languages [Fowler, 2011]. However, these efforts will amortize [Fowler, 2011]. Thus, *Domain Specific Languages* are gaining more and more importance in software engineering, especially in the MDSD field [Strembeck and Zdun, 2009]. Fowler (2011); Stahl et al. (2007) emphasize the improvements of development productivity by providing much more expressive forms for reading and maintaining models. Particularly, the communication between domain experts and stakeholders can be made more efficient and reliable when focussing on functional aspects rather than being bothered by technical details. Consequently, the MDSD community contributed many powerful tools for creating, using, and maintaining DSLs. While tools that use MOF as meta-metamodel are mostly integrated modeling environments, the tools provided under the aegis of EMF with Ecore as meta-metamodel are highly customizable and can be used in any combination. All these reasons make Eclipse a promising tool ecosystem for creating, using, and maintaining DSLs.

To conclude, DSLs unfold the full potential of the MDSD procedure through enabling automatic transformations and providing efficient interfaces between modelers and the model. Furthermore, powerful tools are available and they are accepted in the relevant research fields as well as in industry. Hence, they form the central idea behind this thesis.

2.5 Requirements and Contributions

To successfully solve the problem stated in Section 1.1, an approach is required covering all research contexts introduced in this section. By situating the UsiXML description language (see Section 2.2) into the MDA context (see Section 2.3), Vanderdonckt (2005) shows that this research effort encompasses the two research contexts MBUID and MDSD. MARIA XML can be situated in these two contexts[10] as well, as it is a formal modeling language featuring model transformations. The ACPLT/HMI framework (see Section 2.1) attempts to introduce an HMI development procedure in the automation domain that additionally encompasses both the MBUID and the MDSD context. However, its metamodels were designed

[10] With the exception of situating MARIA XML into the MDA context, as MARIA XML use XSD as meta-metamodel.

closely to the supported platforms, which makes it residing on the FUI level of the CRF (see Section 2.2).

Van den Bergh and Coninx (2005) express the models of MBUID using the MDSD terminology by choosing suitable UML diagrams with which these models can be realized. Additionally, they explicitly map them to the abstraction levels introduced by the MDA specification. The *UML based Web Engineering* (UWE) [Koch et al., 2008], *GUILayout* [Blankenhorn and Jeckle, 2004] and *MANTRA* [Botterweck, 2007] use UML diagrams to model various aspects of user interfaces. These approaches can be completely mapped to the Cameleon Reference Framework. *Wisdom* [Jardim Nunes and Falcão Cunha, 2000] and *UMLi* [da Silva and Paton, 2003] are UML based techniques for modeling user interfaces at a pure functional level covering the CRF only partly. All of these UML based approaches have in common that they either provide a lightweight or a heavyweight extension of the UML specification. Therefore, they constitute a domain specific dialect of the UML specification (see Section 2.4).

To conclude, none of the explored approaches covers all research contexts. Furthermore, none of these approaches has ever been successfully adopted by industry outside of research projects, to the authors best knowledge. This makes it necessary to work out an own appropriate solution with the following requirements to be elaborated on:

(1) A **CUI Modeling Language** needs to be worked out.
(2) This modeling language must provide a **high-fidelity concrete syntax notation**.
(3) A **model verification tool** is necessary to ensure the correctness of models.
(4) **Vertical and horizontal transformations** are required to create runtime solutions for different platforms from a single model.
(5) A **low-fidelity tool interface** should be provided to reuse engineering data from other tools.
(6) **Support for iterative development procedures** such as the user-centered design process must be provided.

The remainder of this section discusses these requirements more detailed.

Requirement 1 — CUI Modeling Language.

Requirement Definition: *A CUI modeling language that provides means for completely expressing its target domain "HMIs for monitoring and operating of technical processes with respect to the field of production automation" is required. They consist of:*

(1) means for graphically visualizing dynamic data, alarms, and trends;
(2) interaction means for operators underlying a haptic input modality (keyboard, mouse, touch devices, and function keys of Industrial PCs);
(3) process data and automation specific communication relationships;
(4) means to include custom functionality and the connection to external programs.

Motivation: The CRF (see Figure 2.2) proposes to abstract operative characteristics from the computing platform in a modeling language situated at the *Concrete User Interface (CUI)* level. Modeling at this level is close to the mental model of the domain expert while still being not faced with decisions about technical realizations.

Deficiencies of existing Solutions: Current visualization systems like *WinCC* and *InTouch* (as discussed Section 2.1) produce solutions that strongly depend on specific platforms through their vendor-specific runtime environments. Particularly, this excludes new generations of devices or embedded ones for monitoring and operating close to the technical process. The UIML modeling language, introduced in Section 2.2, allows to create canonical descriptions of user interfaces that are, however, always targeting specific platforms [Schäfer, 2007]. Thus, it mixes operative characteristics with technical aspects. Other model-based approaches which explicitly treat the CUI level as integral part have serious shortcomings with respect to the requirement mentioned above: UsiXML only defines how to model concrete interactions instead of composing user interface elements [Paternò, Santoro, and Spano, 2009]. MARIA XML allows to describe arbitrary data models using the XSD type definition language, but it does not define interface elements for alarming and trending. Its existing elements lack properties for appropriately visualizing dynamic process data. XIML completely leaves out the concrete level. Moreover, these languages only provide stubs to the underlying application — UsiXML provides an object notation and MARIA XML allows to annotate WSDL documents. Yet, they neither provide means to define communication relationships to automation specific data servers nor allow to extend a user interface description through custom functionality, being similar to the scripting environments of existing visualization systems. Concluding, these existing modeling languages are *Domain Specific Languages* tailored to a specific domain or use case that is not fully compatible with the one of monitoring and operating technical processes.

Contribution 1: Unlike existing visualization systems, the *Domain Specific Language* MOVISA strictly separates operative characteristics from their technical realization thus ensures the sustainability of visualization solutions. While other domain specific languages cannot entirely meet the requirement stated above, MOVISA is tailored to capture the operative characteristics of visualization solutions for supervising technical *production automation* processes.

❊

REQUIREMENT 2 — HIGH-FIDELITY CONCRETE SYNTAX NOTATION.

Requirement Definition: *It is required to provide a concrete syntax notation that is close to the problem being solved, meaning that the modeler should have an idea of the final result already at the time of modeling. It hides the the complexity of models and transformations behind a modeling tool, allowing domain experts to focus on their tasks.*

Motivation: Myers, Hudson, and Pausch (2000) note that model based user interface techniques have not found wide acceptance because they were too abstract. Particularly, domain experts who intend to use the modeling language to create visualization solutions do not know much about models and transformations. With this respect, Streitferdt et al. (2008) identify general needs for better tools: The success of DSLs significantly depends on the expressiveness of their concrete syntax notation [Stahl et al., 2007; Strembeck and Zdun, 2009], as "[h]uman information processing is highly sensitive to the exact form in which information is presented to the senses" Moody (2009). Thus a high-fidelity modeling workbench reduces confusion, lowers the initial hurdles, and raises the efficiency, even when working on huge models.

Deficiencies of existing Solutions: Existing modeling languages, as introduced in Section 2.2 and Section 2.5, are either XML or UML based. Both are rather general purpose notations. Michotte and Vanderdonckt (2008) propose *GrafiXML*, being build atop UsiXML, that allows for modeling user interface elements using a visual editor. However, modeling the application stub and the communication interface requires to edit the plain XML code. The MARIA Tool [Paternò, Santoro, and Spano, 2009] is a visual editor for composing user interfaces based on the

MARIA XML concrete level. Similar to GrafiXML, it forces the modeler to edit plain XML code. Consequently, these modeling tools face developers with complex language notations.

Contribution 2: The MOVISA language contains an efficient concrete syntax notation that combines graphical and textual artifacts so as to provide a very close match to the design problem. All complexity imposed by the core language model is encapsulated behind a high-fidelity modeling interface of the MOVISA modeling workbench. Thus, it fosters domain experts to work and to think in their domain.

REQUIREMENT 3 — MODEL VERIFICATION TOOL.

Requirement Definition: *A tool for identifying modeling errors in early design stages is required.*

Motivation: Preventing modelers to make mistakes is another way of raising the efficiency. It would be naive to believe that there can exist a tool to achieve this. To identify errors in early design stages is an important step towards this direction. Martinie and Palanque (2011) argue that appropriate verification techniques provide ways to ensure the reliability of a system prior to implementation, thus empowering developers by "offering means for reasoning about their system at a higher level of abstraction".

Deficiencies of existing Solutions: WinCC provides a simulation tool in order to verify the correctness of solutions being created with it. While simulating the underlying hardware, the visualization solution can be executed and thus tested. However, everything going beyond syntactical tests performed by the interpreter must be carried out manually. Similarly, none of the existing model based approaches, mentioned in Section 2.2, provides means for ensuring correctness that go beyond the constraints imposed by their metamodel. For instance, XML-based languages lack powerful verification tools: *XML Schemata* only provide simple *key* or *foreign key* relations, as known from relational databases; *Schematron* [ISO/IEC, 2006] goes a step further by making assertions about patterns in XML documents. Even though this forms a sound basis for identifying modeling errors, its range is very limited.

Contribution 3: The MOVISA modeling workbench includes an extensible set of verification rules dedicated to ensure the correctness of models in terms of model integrity checks, as proposed by Raneburger et al. (2011b). Unlike the limited verification means of existing tools, these verification rules can also be expressed using the Java programming language which allows for defining of very comprehensive model constraints.

REQUIREMENT 4 — VERTICAL AND HORIZONTAL TRANSFORMATIONS.

Requirement Definition: *Model transformations are required that automatically generate runtime artifacts of different deployment configurations from models.*

Motivation: Model driven approaches reduce the overall development efforts through well-established and tested transformations, as argued by Streitferdt et al. (2008). Particularly advantageous in the field of industrial automation is that they increase the performance and reliability of generated runtime artifacts. A well-defined and tested set of transformations promises to produce predictable, reproducible, and correct runtime solutions [Vanderdonckt, 2008]. Automatic generation of user interfaces from models fosters usability due to the potential consistency across different devices. Additionally, it reduces the development costs through reuse [Nichols, Chau, and Myers, 2007].

Deficiencies of existing Solutions: Deploying a UIML model to a particular platform means to equip it with an appropriate renderer, which can be considered as a runtime environment. UsiXML features a transformation model that is based on graph transformations. However, this transformation model is part of the user interface model thus containing individual characteristics. In other words, UsiXML allows user interface models for defining its individual behavioral semantics with the consequence of leaving room for interpretations. Both, UsiXML and MARIA XML use XSLT to generate the FUI, which is a pure declarative approach with restricted expressiveness limiting the efficiency of the transformation tool. Furthermore, its too complex syntax is difficult to read and to maintain, even by experienced developers.

Applying one user interface to another context of use thus deploying it to another hardware platform using a horizontal transformation (translation) probably

entails rearranging user interface elements if e.g. the target platform features a smaller display. The *MuiCSer* [Meskens et al., 2009] process framework allows for defining UIML based models for single devices. Constraints defined on these models stipulate in what extent they can be altered to still remain usable. Based on this, an adaptation engine is responsible for tailoring the UI to particular devices at runtime thus probably resulting in an unpredictable user interface behavior.

Contribution 4: MOVISA makes explicit a precise *Language Behavior* definition being the basis for automatic transformations. MOVISA modeling workbench provides vertical transformations deploying a MOVISA model to different computing platforms. It features horizontal transformations to adapt a user interface to another hardware platform. For this purpose, it pursues a design time evolution strategy enabling domain experts to control the generated results. Both vertical and horizontal transformations are combined in an efficient and extensible transformation tool consisting of declarative and imperative languages.

❊

REQUIREMENT 5 — LOW-FIDELITY TOOL INTERFACE.

Requirement Definition: *It needs to be investigated if engineering data can be reused from earlier engineering phases.*

Motivation: Engineering data of previous development phases during automation system engineering, as discussed in Section 2.1, can be reused for the development of visualization solutions. For instance, parameterization data of PLCs can be reused for configuring the process communication relationships.

Deficiencies of existing Solutions: Basically, *Excel spreadsheets* are used to transfer data from engineering tools to visualization systems. WinCC e.g. provides interfaces for an automatic import. However, besides being highly error-prone, this method causes a break in the tool chain due to manually creating and maintaining those spreadsheets.

Contribution 5: To bridge the gaps in existing tool chains, a transformation based framework has been worked out. It uses the data of engineering tools and populates MOVISA models with it through the dedicated *Low-fidelity Tool Interface*. With this, the MOVISA modeling workbench reduces the overall engineering effort and provides a continuous tool chain.

REQUIREMENT 6 — ITERATIVE DEVELOPMENT PROCEDURES.

Requirement Definition: *In order to ensure safe process operations through the generated visualization solutions, the modeling language must be incorporated into incremental and iterative design processes such as the one proposed by Zühlke (2004).*

Motivation: Sheridan (2000) states that poorly designed automation systems increase the load on the human operators. As a consequence, this leads to more human errors and with it perhaps to a reduced quality of the product to be produced through the technical process. Even worse, it could result in an impairment of the safety. VDI/VDE (2005) suggest to establish an iterative, user-centered design procedure in order to resolve "any conflicts arising from deviations from the familiar situation". In the industrial automation context, Zühlke (2004) proposes the *Useware Engineering* process to iteratively design usable *Human Machine Interfaces* by involving end-users in each design phase. Martinie and Palanque (2011) require user interface design tools to support incremental and iterative prototyping activities by encouraging developers to modify their prototypes.

Deficiencies of existing Solutions: User-centered design procedures are about continuous improvement of user interface prototypes, already in early development stages. In MBUID, these modifications can take place at a concrete level where the UI is getting more concrete. (For instance, the Useware engineering process uses UIML at the concrete level.) However, altering UI models at the concrete level might not influence the abstract models, which leads to model inconsistencies. Solving these inconsistent states requires *Model Synchronization* means, which are not part of the Useware process. Hence, necessary modifications are performed always at the abstract level and populated to the concrete ones, leading to unnecessary efforts.

Contribution 6: To enable iterative design processes, a transformation based framework was built around the MOVISA modeling workbench, exploiting the *low-fidelity tool interface*: Vertical refinement transformations create MOVISA models from more abstract user interface models, horizontal update transformations allow for improvements, e.g. after end-user feedback, and vertical synchronization transformations enable to remain all models in a consistent state. A trace model forms the core of this framework.

Chapter 3

Language Model

The *Language Model* forms the core of a DSL, as Figure 2.5 illustrates. This chapter describes the deduction of the *Language Model* — the metamodel — by analyzing the *Target Domain* as a fundamental requirement to abstract this piece of the "real world" into modeling elements and to establish relationships between them, both expressed within the *Core Language Model*. Characteristics of the target domain, that cannot be expressed in the core language model will, be made explicit using suitable *Language Model Constraints*. Both the core language model and the constraints defined on it determine *what* models have to capture, namely the *variable operative characteristics*. The *Language Behavior* defines *how* the language works. Thus, the knowledge that is expressed in the *Language Behavior* is identical for every application which complies with the definition of *fixed operative characteristics* (see Definition 2.5 on page 14).

3.1 Analyzing the Target Domain

A key factor to success of a domain specific modeling approach is that the modeling language captures the target domain very precisely. Yet, it sets a clear boundary for its intended use [Kelly and Pohjonen, 2009; Stahl et al., 2007]. As the DSL to be created is intended to capture the operative characteristics of visualization solutions, existing tools to create these solutions will be investigated in terms of their configurable parameters.

The following process visualization systems were the basis for this investigation: WinCC, InTouch, Genesis64, and ProWin. They are commercial and mature visualization systems, producing solutions for the application domain *Production Industries*. Automation engineers can make use of powerful tools (see Section 2.1) to create and to configure visualization solutions for monitoring and operating a particular technical process. Those visualization solutions have to support the available communication protocols to gather process data. This data is appropriately presented to operators through a *Graphical User Interface* (GUI) containing input

devices (controller, comp. Figure 1.1) and output devices (graphical display, comp. Figure 1.1). Additionally, this provided functionality can further be customized using scripts. Since creating a domain specific modeling language requires a good understanding of the problem domain [Kelly and Pohjonen, 2009], the following sections explain relevant properties of visualization solutions. They will be addressed as *operative characteristics*. Specific requirements are made explicit as basis for deducing the *Language Model*.

3.1.1 Scripting Language

Using a scripting language, visualization solutions can be customized with additional functionality. WinCC supports both VBA[1] or C scripts to dynamize graphical objects, handle data values, read and write process data, or even to plan jobs such as printing daily protocols. InTouch allows to start arbitrary applications, simple calculations, and reading or writing process data using an own BASIC like scripting language. Visualization solutions operated by Genesis64 can be customized using either VBA or JScript[2]. ProWin comes with its own scripting language.

Each visualization product comes up with slightly different methods to execute the required series of instructions encapsulated in these scripts. Generally, they define triggers that are sensible to the following events:

Time: Scripts are executed either in an acyclic manner (a period of time has expired or a predefined point in time has been reached) or in a cyclical one (with a given interval).

User Interaction: Execution of scripts is triggered by users.

Change of Process Values: Scripts can be configured to be executed if the value of a particular process variable changed.

Window/Application: Opening or closing the application (or single windows) triggers execution of scripts.

Summarizing, due to the diversity of industrial production processes, it is not worthwhile to provide standard building blocks for each foreseeable requirement. Hence, visualization systems provide customization means based on limited scripting languages. The aforementioned kinds of events act as activators of these scripts.

[1] *Visual Basic for Applications*

[2] JScript must not be confused with JavaScript. JScript is a Microsoft scripting language for the *Internet Explorer*.

3.1.2 Process Data

Process visualization solutions gather data from field devices (PLCs) to genuinely reflect the actual process states. They write data in form of set points to these devices for operating purposes. For this reason, a visualization solution contains a data model that organizes *process variables*, underlying a strong data type system. Their values make user interface widgets dynamic. Modifying these process variables causes state transitions of the technical process. To ensure the communication between this data pool and the technical process, Zacher and Wolmering (2009) distinguish between *direct* communication relationships, using specific field bus protocols such as *ModbusTCP*, and *indirect* communication via a specific middleware such as *OPC*[3] or *FCML*[4] [Bry et al., 2008]. Every communication protocol is, among other things, characterized by an individual data specification.

Depending on the particular plant infrastructure, a visualization solution has to handle different communication protocols and, thus, several data specifications. For this purpose, it typically provides an architecture that is based on individual communication drivers as depicted in Figure 3.1 [Zacher and Wolmering, 2009]. Each driver is able to handle a specific protocol. It consists tailored process variables that contain data specification specific parameters. As a result, configuring both the communication relationships and process variables is always accomplished by using the particular terminology of the communication protocol.

As process variables represent measurement points of sensors, they indirectly provide information about exceptions or deviations. Limit values of process variables are defined at time of engineering. At runtime, an alarm management system observes the actual values and raises alarms when the defined limit values have been exceeded. Current visualization systems allow to group alarms according to their belongings, such as the location of the problem, and might filter them. Furthermore, process variables also give information about the quality of the product to be produced: For provisioning this data, legal regulations exist particularly for products such as food or drugs. Hence, current visualization systems enable to

[3]The *OPC Data Access* specification that is based on Microsoft's DCOM technology has gained significant importance in automation. A current trend is to use vendor neutral protocols based on web services with *OPC XML-DA*. The recent OPC specification *Unified Architecture* provides both a web service based data exchange and high-performant binary TCP connections [Lange, Iwanitz, and Burke, 2010; Mahnke, Leitner, and Damm, 2009].

[4]Similar to OPC, the *Facility Control Markup Language* (FCML) provides a middleware for a uniform data exchange between SCADA applications and devices, both being connected using different protocols. It introduces its own infrastructure and an XML based communication protocol.

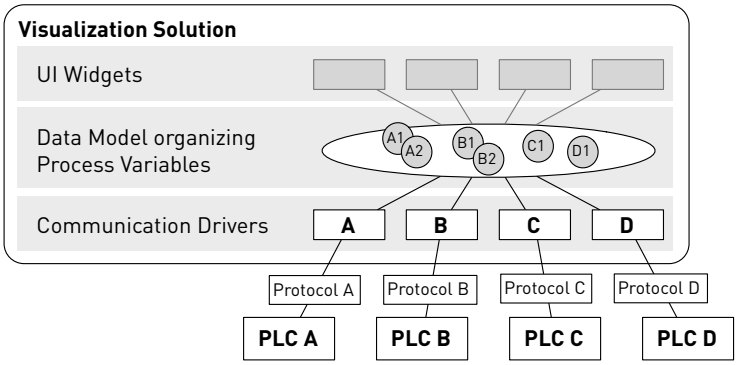


Figure 3.1: Principle of providing communication drivers for different protocols: A data model organizes different types of process variables; either the data model or the process variable is responsible for converting the data to be used, e.g. by user interface widgets.

configure which process values need to be logged for future analysis.

Based on the aforementioned facts, visualization solutions can be deployed in various ways. Figure 3.2 shows a Client-/Server architecture as introduced by Tauchnitz and Maier (2004, p. 220 and p. 218, Bild 1): Both stationary operating stations and web clients are connected to an OPC XML-DA Server. It acts as middleware and is responsible for managing process data.

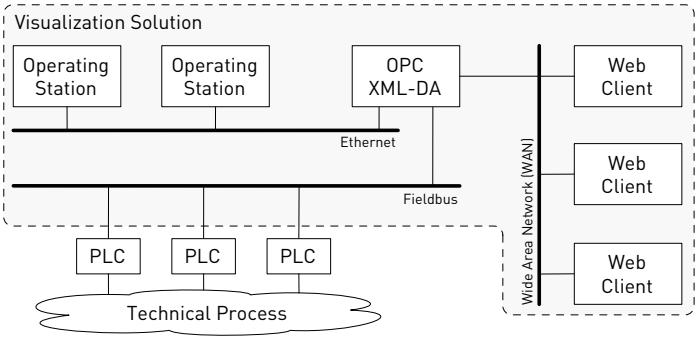


Figure 3.2: Exemplary deployment of a visualization solution: Client-/Server architecture using OPC XML-DA middleware for providing process data also to web clients.

To sum up, visualization solutions support several communication protocols, thus, being able to handle *process variables* of different data specifications. This includes to read and to write these process data, even if a conversion to raw sensor

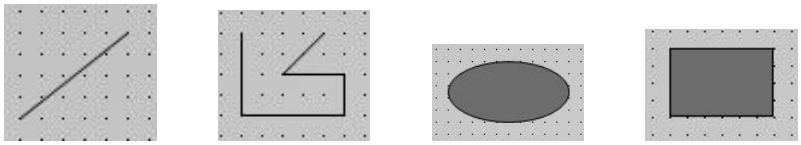
values is necessary beforehand. Intervening in the process requires to write several process variables, then this write operation needs to be atomically. (The OPC XML-DA specification e.g. provides the *Write List* method and ModbusTCP allows to write complete address ranges at once.) Finally, a visualization solution needs an alarm management system and it must be configurable which data are to be archived.

3.1.3 User Interface Components

Operators are in charge of supervising a process to produce particular goods with the expected quality. Therefore, they need to be informed about the actual process states. They also need to intervene in the process by entering set points through a user interface. To successfully achieve this, VDI/VDE (1999) recommends to structure the information as hierarchical mimics on graphical displays, providing a simplified representation of the plant infrastructure, and to define reasonable and easily accessible navigation means between these displays. The hierarchy typically ranges from an overall plant display over area displays up to detail displays.

For establishing mimics, the investigated visualization systems offer appropriate, predefined user interface components — the so-called *widgets*. A distinction is made between the following kinds of widgets:

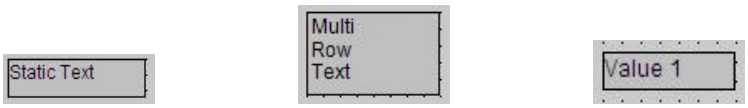
(1) *Basic geometrical objects* as depicted in Figure 3.3.



(a) Line (b) Poly Line (c) Ellipse (d) Rectangle

Figure 3.3: Basic geometric user interface widgets.

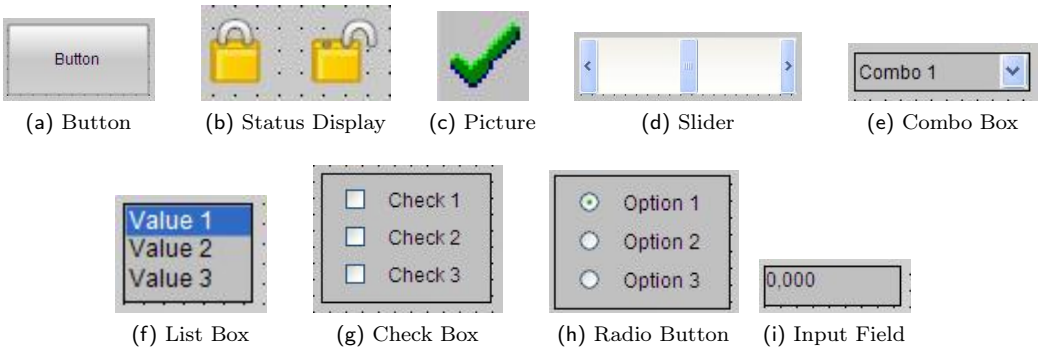
(2) *Common interaction widgets* are well-known from typical office applications. *Text* widgets (see Figure 3.4) are used to deliver single-line or multi-line text. A *Text List* widget shows one of many configured texts depending on a value. *Button* widgets (see Figure 3.5a) are used to trigger actions, e.g. to execute a script (comp. Section 3.1.1). Additionally, buttons can also be configured to have a state (see Figure 3.5b, also shows that buttons can have specific pictures). Figure 3.5c shows a dedicated *Picture* widget containing static or dynamic images. *Slider* widgets (see Figure 3.5d) enable to enter set points seamlessly within a defined range. *Combo Box* widgets (see Figure 3.5e),



(a) Text (b) Multi Row Text (c) Text List

Figure 3.4: Text widgets.

List Box widgets (see Figure 3.5f), *Check Box* widgets (see Figure 3.5g), and *Radio Button* widgets (see Figure 3.5h) are intended for presenting several options to operators they can choose from. (List box and check box widgets allow to choose one or many options. Combo box and radio button widgets only allow to choose a single option.) *Input Field* widgets (see Figure 3.5i) enable operators to enter arbitrary data such as set points. As this entered data has an impact on the process, not only does type safety have to be ensured but also that the data is within a defined range of values.



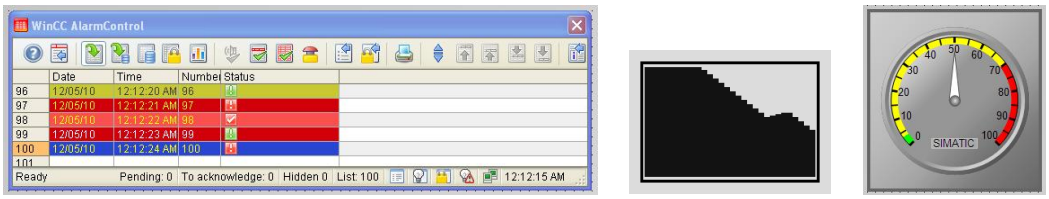
(a) Button (b) Status Display (c) Picture (d) Slider (e) Combo Box

(f) List Box (g) Check Box (h) Radio Button (i) Input Field

Figure 3.5: Common interaction widgets (also known from typical office applications).

(3) *Automation specific widgets* are mandatory for monitoring and operating technical processes: As operators must instantly be informed about exceptions or even deviations in the process, VDI/VDE (1999) requires each window of a visualization solution to be equipped with an *Alarm Control* widget (see Figure 3.6a). Every time, it shows the most recent alarms (comp. Section 3.1.2) together with required meta information such as severity level, time of occurrence, alarm state, help texts, and comments.
Operators are required to make the right decisions while operating the technical process. Therefore, they potentially need to compare actual process values with past ones and additionally need information about the development of particular process values. In order to ensure this, two widgets were established:

The *Trend Chart* widget (see Figure 3.6b) allows to monitor the development of process values with a particular resolution within a certain period of time. A distinction is made between real time trends and historical trends: Real time trends store process values only within the configured period of time. Historical trends archive forever. Hence, historical trends need appropriate data servers (comp. Section 3.1.2). Rapidly changing values with no need for recording past values are usually monitored using *Gauge* widgets (see Figure 3.6c). This dial instrument reflects the velocity and direction of the change in process values.



(a) Alarm Control (b) Trend Chart (c) Gauge

Figure 3.6: Interaction widgets that are mandatory for monitoring and operating technical processes.

(4) *Complex widgets:* If a specific symbol configuration, a particular combination of several widgets, or entire mimics are often used in projects, so called *HMI Typicals* allow to reduce the development effort. These typicals are widgets of any complexity being available through libraries. The investigated visualization systems provide means to integrate arbitrary ActiveX components.

Summarizing, visualization solutions consist of graphical displays representing the hierarchical structure of a plant and define reasonable and easily accessible navigation means between these displays. Each graphical display is comprised of different classes of user interface widgets, as discussed above. Furthermore, component libraries foster reuse and consistency thus they are required by practitioners (comp. [Völter, 2009]). Widgets of each class are characterized by a collection of configurable properties that can be classified by (1) those defining the initial appearance of the user interface (*Representation Properties*), (2) those defining how the widget behaves according to actual process data (*Animation Properties*), and (3) those defining how an operator can interact with the particular widget (*Interaction Properties*). The following sections give a detailed over view of these classes of properties.

Representation Properties

User interface components can be configured by a set representation properties defining their initial appearance. Table 3.1 presents identified representation properties.

Table 3.1: Classes of common properties that were identified during investigating visualization systems as well as running solutions.

Property Class	Characteristics
Alignment	Widgets that contain text expect the horizontal and vertical alignment of the text to be configured.
Border	Used to assign a border to a widget. Various parameters, such as *Width*, *Color*, or *Style*, can be set.
Color	Contains parameters to configure a widget's background and foreground color, including specification of a transparency.
Flash	A particular event or process state might require the operator's special attention. Hence, a widget's border or color properties can be configured to flash.
Format	This class of properties allows to configure the appearance of text by defining, among others, *Font Size*, *Font Family*, and *Font Style*. (Font color is configured using foreground color of *Color* class.)
Operation	Determines if a widget is accessible by an operator.
Position	Defines the exact position of the widget on the screen using X, Y, and Z coordinates.
Rotation	Using these parameters, a specific angle can be configured to widget. This leads either to more powerful representations or can be used to call the operator's attention to a specific region.
Size	Defines the size of the widget in terms of *Width* and *Height*.
Tooltip Text	Configures a text that will be shown if the widget is touched with an interaction device (e.g. by moving the mouse pointer across the widget).
Visibility	Defines whether the widget is visible or not.

Animation Properties

To keep operators always informed about actual process states, suitable dynamic properties are required. Existing visualization systems allow all properties (listed in Table 3.1) to be altered according to actual process values or by means of scripts. For instance, a process value might alter the *Height* property of a rectangle widget (see Figure 3.3d) to represent the fill level of a tank. Table 3.2 shows representative

dynamic property classes.

Table 3.2: Various possibilities to alter widgets.

Property Class	Characteristics
Value Output	Using this property, a text widget might be configured to show the actual value of a process variable.
Size	Depending on a process value, the size of a widget can be altered.
Position	A widget varies its position according to a process value.
State	A process value might alter the following properties: *Operation*, *Flash*, *Visibility*.
Navigation	A particular process state might cause to show another window.
Appearance	*Color* or *Text* properties can be altered.

Altering widget properties can also be bound to certain conditions. This forms a powerful instrument to block individual widgets for operation or to call the operator's attention to a specific region of the screen. For instance:

> CHANGE widget's background color FROM blue TO red
> IF process value IS GREATER THAN 230.

Interaction Properties

Monitoring and operating technical processes also means to interact with the user interface. Hence, the widgets of a visualization solution define *how* an operator can interact with the widget and *what* effect this interaction has. The *how* is limited by the platform capabilities (mouse, keyboard, or function keys) to *Click*, *DoubleClick*, *Touch*, and *Keystroke*. The *what* has a wide range of possibilities, depending on the visualization system. Table 3.3 gives a brief overview of general effects (*what*).

Table 3.3: Configurable interaction effects.

Interaction Effect	Characteristics
Read/Write Process Data	Causes the application to read or write process variables.
Execute Script	User triggered execution of a script.
Execute Application	An arbitrary application, such as Microsoft Excel, can be executed.
Navigation	A user causes the application to activate another window.
Printing	The current screen can be sent to a printer.

3.2 Core Language Model

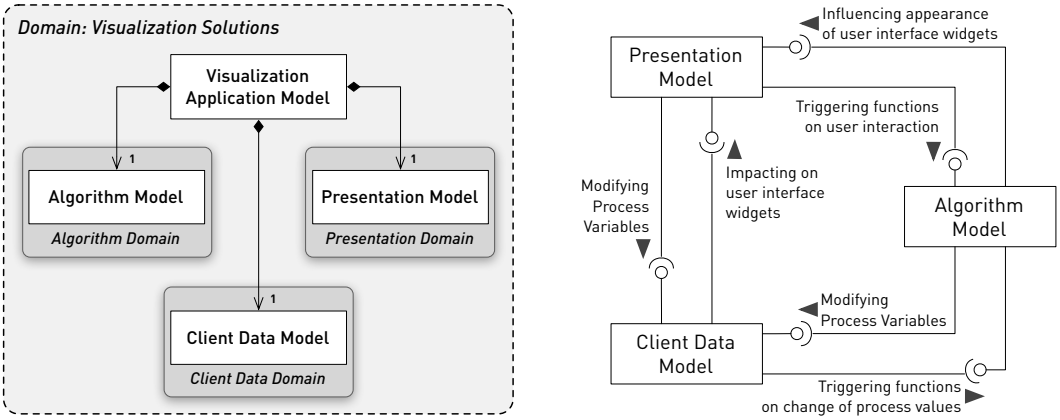
Summarizing the investigation of the target domain in Section 3.1 implies three fundamentally different kinds of concerns to be addressed by the *Core Language Model*: *Scripting Language*, *Process Data*, and *User Interface*. When formalizing the domain characteristics into a metamodel, these concerns become individual subdomains of the system, each being characterized by different abstractions and possibly different notations. Stahl et al. (2007) indicate that dividing a domain into subdomains fosters handling of complexity and enables mixing of concrete syntax notations. Völter (2009) additionally emphasizes the importance of reasonably chosen subdomains. They form viewpoints on a system fostering to describe different aspects of the systems by different roles at different times, e.g. when collaborating in teams. The following subdomains will be dealt with individually:

(1) *Algorithm Domain.* Adding custom functionality, that cannot be realized using provided components, is achieved through limited programming languages. These custom functions range from manipulating process data and performing calculations up to influencing the appearance of the user interface. A user interaction, altering process variables, or timer events might trigger these functions.

(2) *Client Data Domain.* Reading and writing process variables is based on communication relationships using various protocols and different data specifications. Actual values of process variables have an impact on the appearance of user interface and form the foundation for an alarm management, requiring the definition of limit values. Process values can be recorded for assessment.

(3) *Presentation Domain.* Predefined elementary and complex user interface widgets, as introduced in Figure 3.3, 3.4, 3.5, and 3.6, are configurable with *Representation* (Table 3.1), *Animation* (Table 3.2), and *Interaction* (Table 3.3) properties. Complex widgets enable reuse in other projects.

Figure 3.7a illustrates the interrelationship between the subdomain and the resulting modeling language. The element Visualization Application Model, representing the target domain, forms the root element consisting of the submodels Algorithm Model, Client Data Model, and Presentation Model. Each constitutes a subdomain. Figure 3.7b shows the required interfaces as connection points between the individual submodels as identified above.



(a) The target domain has been divided into reasonable subdomains each coming up with a particular submodel.

(b) Required interfaces between submodels in order to specify interrelations of the different components of visualization solutions.

Figure 3.7: The root elements of the *Core Language Model* and their main relationships.

3.2.1 Algorithm Model

Platform independence of a *Scripting Language* can be achieved by using a standardized language. Following this approach, *Geneses64* integrates *JScript* that requires, however, a specific operating system (comp. Section 3.2.1). Equipping each platform to be supported with a suitable runtime interpreter might not be possible on embedded or mobile platforms. Furthermore, integrating a scripting language into a model might cause other difficulties, because they have different and, certainly, incompatible (meta-)metamodels. Consequently, interrelations with the other submodels can only be defined informally, violating the requirements of MDSD.

An abstract and computationally complete software specification or modeling language, such as *Executable UML*, is capable of overcoming these shortcomings by addressing the custom functionality without making any decisions about platform specific concerns. The FUML specification [OMG, 2008] narrows down the UML specification [OMG, 2010] to an executable subset and defines precise execution semantics (comp. *Language Behavior*, Section 2.4). Regardless of that, Mellor and Balcer (2002) introduce three basic projections on the UML metamodel to gain computational completeness: (1) *Classes* abstract the real world as objects, attributes, and relationships; (2) *State Machines* define a lifecycle for these objects; and (3) *Procedures*, being composed of *UML Actions*, establish the particular states.

Based on these considerations, three projections on the UML metamodel, as proposed by Mellor and Balcer (2002) and following the FUML specification OMG, 2008, were introduced (see Appendix B) to realize such platform independent scripting language as discussed in Section 3.1.1. According to the UML specification [OMG, 2010], *signals* are the means for intra- and inter-model communication. Consequently, they will be used to establish the interfaces between the submodels[5] depicted in Figure 3.7b. Nevertheless, introducing additional concepts was necessary for the sake of interacting with both, the *Client Data Model* and the *Presentation Model*. In the following, these extensions are explored in detail.

Type System

Very complex data structures can be built using the type system of UML, represented by the modeling entity DataType [OMG, 2010, p. 62]. However, UML's data type system only has an imprecise and informal description of Primitive Types[6], as it does not define concrete subtypes. On the other hand, data server specifications (comp. Section 3.1.2) rely on precise descriptions. Therefore, concrete primitive subtypes, as defined in *Tutorial D*[7] [Date and Darwen, 2007], were added. Furthermore, a Structured Type was made explicit in order to map automation specific data types, such as the ones defined by *OPC UA* (comp. Section 3.2.2), more precisely. Figure B.5 shows the resulting type system.

Boundary

Neither UML nor FUML include means to address platform related concerns [Starr, 2002], such as time, persistency, or converting raw sensor values, as identified in Section 3.1.2. These decisions will be made by the model compiler [Mellor and Balcer, 2002], the transformation. However, if modeling entities need to interact with these platform related concerns, their realization cannot be left to the model compiler. It must be part of the model. For instance, when executing custom scripts periodically, as introduced in Section 3.1.1. Even though the required timer is realized using platform specific source code, the model needs a reference to this timer to define when to execute which procedures. In case of converting raw sensor values, a reference to the respective data source in the model is required as well as to the consumer of the converted value.

[5] With the exception of the two interfaces "Modifying Process Variables", they use custom build UML actions.

[6] A primitive type has no relevant substructure [OMG, 2010, p. 127].

[7] Tutorial D is an abstract language specification of relational database query language.

Fowler (2011) proposes two possible solutions to describe something that is beyond the capabilities of the language: (1) Extending the DSL with these specific aspects is the first solution, but it can lead to a significant complexity. (2) "Embed some foreign code into an external DSL to provide more elaborate behavior than can be specified in the DSL." [Fowler, 2011, p. 309]. For the latter solution, the model needs well-defined extension points so that both, the model and the source code, can interact. In the field of engineering distributed control applications, the specification *IEC 61499* [DIN, 2006] defines a component-based and generic modeling approach using function blocks. In order to include vendor specific aspects, it proposes the use of proprietary *Service Interface Function Blocks* [Frey, 2008]. Even though the idea behind that concept is to provide interfaces to external functions within the modeling language, the realization is very similar to extending the language.

Consequently, the modeling entity Boundary was introduced to provide well-defined extension points. Figure B.4 depicts that the Boundary contains two types of interfaces, the Required Interface and the Provided Interface that are both responsible to Signals. Figure 3.8 illustrates the basic principle: The Required Interface defines Signals that can be received and will then be processed by the system behind the Boundary. The Provided Interface defines Signals that will be sent by the system behind the Boundary. Consequently, the Boundary can completely be mapped to the UML Component type. However, while a Component features a Component Realization, the system behind the Boundary must be realized using any platform specific source code, even though it is part of the model. In this way, it additionally allows for taking advantage of legacy code libraries, as suggested by Selic (2003). In terms of executing a script periodically, the Boundary only defines a Signal in the Provided Interface that can be used to trigger transitions of a state machine.

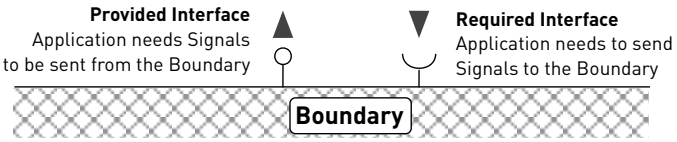


Figure 3.8: Basic principle of the Boundary concept.

Additional Actions

Additional Actions to establish the procedure of States were necessary so as to integrate the Boundary and to realize the interface "Influencing appearance of

user interface widgets" (see Figure 3.7b) to the Presentation Model. In both cases, signals need to be sent using a Send Signal Action [OMG, 2010, p. 229, Figure 11.4]. A Send Signal Action, though, requires the definition of a target, where the signal should be sent to (comp. Figure B.12). For this purpose, UML provides suitable actions to retrieve those targets (e.g. a Read Self Action returns the host object of an action). Similarly, both, a Read Boundary Action (see Figure B.14) for addressing Boundaries and a Read Presentation Model Action (see Figure B.15) for addressing the Presentation Model, were introduced.

A different consideration for the interface "Modifying Process Variables" (see Figure 3.7b) is required: Signals are well-known as asynchronous communication means. However, writing data items has usually to be done in a precisely defined order. Although an appropriate language behavior definition might constrain signals to be executed synchronously, modelers — usually domain experts without metamodeling experience — might be confused or at least feel uncomfortable using signals. On the other hand, Requirement 2 on page 24 imposes to enable modelers to work and think in their domains. Consequently, both, a Read Data Item Action and a Write Data Item Action, are provided by the metamodel, as depicted in Figure B.18. They can be connected using Control Flows [OMG, 2010, p. 366] to establish a serial and synchronized read or write operation. When using a Fork Node [OMG, 2010, p. 387], parallel operations can be realized.

3.2.2 Client Data Model

Concluding Section 3.1.2, the values of process variables — possibly underlying different data server specifications — represent measurement points in the plant facilities. Thus, the pool of these process variables captures the operative states of the technical process. User interface widgets (to be provided by the Presentation Model, see Section 3.2.3) together with the Actions of the Algorithm Model appropriately prepare these data for human operators. This requires an information exchange between field devices (e.g. PLCs) and the visualization solution, as Figure 1.1 illustrates.

Exchanging information between a visualization solution and devices of plant facilities requires, according to Eberle (2007), sender and receiver to rely on the same communication model. However, they might use different information models and, thus, define different points of view on the exchanged information. Characteristics of communication models are, among others, the knowledge about the physical information representation, such as messages, protocols, services, and infrastructures. These characteristics are *fixed operative characteristics* and,

thus, identical for all visualization solutions. Capturing them is beyond the responsibility of the Client Data Model. This knowledge will be added to it through transformation. Information models, on the other hand, are formed by the pool of process variables. Each process variable is characterized, among others, by a data type, an address in the namespace of the data provider, and meta information such as a name. Consequently, each visualization solution requires an individual information model that is characterized by the given technical process and needs to be configured in the Client Data Model.

Figure 3.9 illustrates the main concept of the Client Data Model. Configuring process variables requires to capture certain aspects of the information model of each relevant data provider. For this purpose, the Client Data Model provides the Technical Data Perspective defining modeling elements Server and Server Data Item. Configuring these elements accompanies with addressing the data to be exchanged in the namespace of the particular data server. Especially as the Client Data Model has to support different data specifications, a unified view on these different information models is required to provide a single interface to access data of different data servers. This is achieved by the Logical Data Perspective. Basically, it defines the modeling entity Data Item representing a logical process variable on which the Presentation Model and the Algorithm Model operate. The information mapping between both perspectives is subject to the transformation, as the required knowledge belongs to the fixed operative characteristics. Figure B.60 formalizes this concept in a metamodel, whereas Figure 3.10 gives an illustrative view on it.

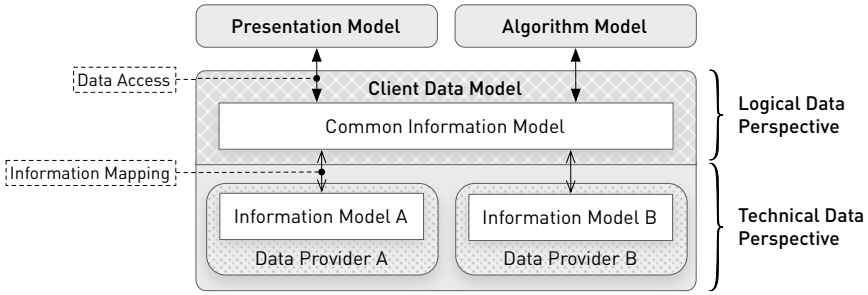


Figure 3.9: A common information model provides a unified interface to different data provider specific information models.

Figure 3.10 shows that the idea of separating both types of information models into logical and technical perspectives accompanies with the approach followed by existing visualization systems (comp. Section 3.1.2): They provide communication

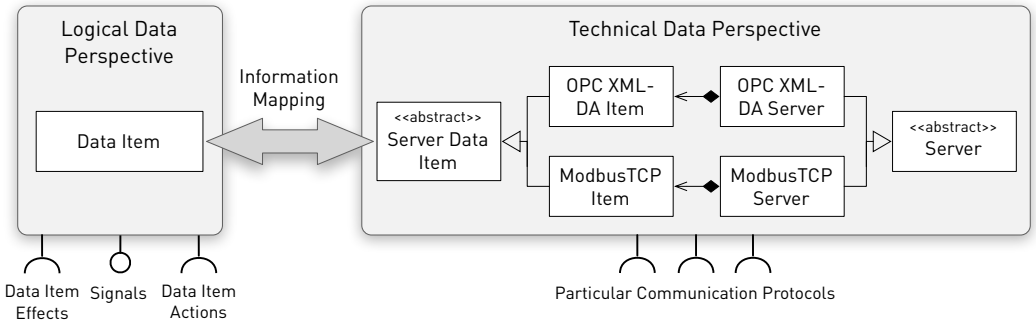


Figure 3.10: The Logical Data Perspective represents the common information model, the Technical Data Perspective captures the data provider specific information models. Both perspectives are connected through an information mapping.

drivers for each data specification (see Figure 3.1). The following sections give a more detailed survey of the individual characteristics of the Client Data Model.

Technical Data Perspective

Gathering data is accompanied with addressing the demanded pieces of information in the information model of the particular data provider. For this purpose, a precise configuration of this data is required. However, the data specifications used in industrial automation are so diverse that a reasonable abstraction of these specifications in a single generic data structure is not possible. Additionally, these abstraction processes come with the removal of information, which is required at transformation time. Thus, the requirement of completely capturing the target domain cannot be met. Consequently, the Technical Data Perspective of the Client Data Model provides, as depicted in Figure B.64, for each data specification a suitable modeling element Server that contains specific Server Data Items. These items represent process variables in the information model of the particular data provider. In this way, configuring the required parameters to address these items can be done using its characteristic terminology. Table 3.4 introduces the characteristic parameters of representative automation specific data models.

Table 3.4: Characteristic parameters of different data server specifications.

Parameter	Description
OPC XML-DA (Figure B.65)	
Item Path	OPC XML-DA defines a tree-like information model, as explained by Lange, Iwanitz, and Burke (2010, p. 39). Addressing individual nodes in this tree requires the *Path* to and the *Name* of the node to be returned.

Parameter	Description
Item Name	
Item Type	OPC XML-DA uses XML standard types.
ModbusTCP (Figure B.66)	
Base Type Reference Address	ModbusTCP is a register based data model that defines four base types (bit-wise read/write, register-wise read/write). To each base type, 65.536 addresses can be assigned. Therefore, addressing individual data always requires the tuple $(BaseType, ReferenceAddress)$.
Type	ModbusTCP does neither define any semantics of the data nor specific data types. Hence, the client application needs to know how to interpret the byte stream.
OPC UA (Figure B.67)	
Browse Path Browse Name	Although OPC UA provides an object-oriented based information model, accessing individual objects occurs in a tree-like fashion, which is very similar to OPC XML-DA.
Type	OPC UA provides a comprehensive type system. It first distinguishes between ordinary Data Variables carrying measured values and Properties carrying meta data of the server (comp. Figure B.67). Both object types contain a Data Type that is very similar to the ones defined in *Tutorial D* (comp. Section 3.2.1 and [Date and Darwen, 2007]), as depicted in Figure B.68 and Figure B.69.

Similar to the investigation on the data specifications shown in Table 3.4, other data providers can be integrated by adding appropriate Server and Server Data Item elements to the Client Data Model as long as the particular information models allow to access process variables. However, always extending the metamodel to support another data specification has two significant drawbacks:

(1) the metamodel can gain a complexity that could be hard to handle,
(2) an existing tool environment needs to be extended, too.

Consequently, a Generic Server element was introduced (comp. Figure B.71). While it allows to simply include new data specifications, it requires to extend the transformation rules to generate the necessary communication routines as well as the definition of validation rules (comp. Section 3.3) to keep the models formal.

Concluding, the extensible adapter-like structure, defined within the Technical Data Perspective, enables modelers to configure process data and communication relationships with the terminology provided by the particular data server specifica-

tion. However, the diversity of available data specifications demands a compromise between complexity and formality.

Logical Data Perspective

Data Items, representing logical process variables, are the core components of the Logical Data Perspective — the logical information model. Basically, a communication relationship can be narrowed down to the operations *read* and *write*. A special case of a read operation is a *subscription*. (A data consumer intimates to get informed by the data provider only if the agreed data have changed.) While a read and write operation is triggered manually, e.g. using a Write Data Item Action (comp. Section 3.2.1), a subscription needs to be configured using the dedicated modeling elements Subscription and Monitored Item (see Figure B.61). By this means, both, the sampling rate for observing a process variable and the interval for publishing new values, can be accurately defined.

VDI/VDE (2005) requires presenting information that ensures unambiguous interpretation. For instance, "[t]he unit following a value makes the value definite" [VDI/VDE, 2005]. Configuring a unit to a value can be subject to the particular user interface widget that presents this value to a human operator. However, as a value might be presented on several mimics, it reduces configuration efforts when Data Items ensure its correct interpretation. For this purpose, the modeling element Dimension is provided by the Logical Data Perspective (comp. Figure B.61). A Data Item has to refer to exactly one Dimension to get a particular meaning. Moreover, a Dimension can be used for several Data Items for the sake of reuse. Additionally, the [VDI/VDE, 1999] requires that values to be compared should have the same number of digits after the decimal point. Configuring this characteristic is subject to the Dimension as well.

Data Items represent process variables. Thus, they are connected to measurement points or actuators of the technical process. To ensure safe operations, a Data Item provides the following properties:

MinValue/MaxValue: These properties define a range of values for the expected set point values. (Comp. Section 3.1.3 where the input field in Figure 3.5i requires a valid data range to be configured.)

OwnedAlarmBehavior: This property allows for defining an arbitrary number of limit values to be used for configuring an alarm management (comp. Section "Alarm Perspective"). Alarm Behaviors specify events that require an intermediate reaction of the operator. Such events happen if the respective process variable reaches a certain value. However, Alarm Behaviors do not

define the particular kind of event to occur, as it is subject to the configuration of an alarm management, as discussed in Section "Alarm Perspective".

Data Items, as the central concept in the Client Data Model, are further on the means for establishing the interfaces "Impacting on user interface widgets" to the Presentation Model and "Triggering functions on change of process values" to the Algorithm Model (see Figure 3.7b). For this purpose, they define the relationship Signal On Change (see Figure B.60).

By means of the Points To relationship (see Figure B.60), several Data Items might refer to a Server Data Item. Hence, one process variable can e.g. be used with different dimensions. Resolving this relationship is subject to the transformation. Section "Information Mapping" gives more details about this information mapping. If a Data Item lacks this Pointing To relationship, it was configured as local process variable and is able to temporarily store values for realizing custom functionality through the means provided by the Algorithm Model.

Information Mapping

Only *fixed operative characteristics* determine a mapping between both the Server Data Items of the technical perspective and the Data Items of the logical perspective. Therefore, the transformation rules capture exclusively the required mapping knowledge. In order to ensure a correct mapping and, thus, the generation of reliable runtime artifacts, both, the logical process variables and the technical process variables, need to be type safe. Data Items of the logical perspective use the type system provided by the Algorithm Model (comp. Section 3.2.1). Server Data Items are characterized by a data server specific type system. Checking if both type systems can be mapped on each other can be the subject to the transformation. However, according to Requirement 3 on page 25, modelers need to be informed about possible type mismatches at an early stage of the modeling process. This can be ensured by formulating appropriate *Language Model Constraints* (see Section 3.3).

Alarm Perspective

Alarm Management systems aim at safe operation of automation facilities by appropriately informing human operators about deviations in the technical process. They foster the availability of automation devices, prevent losing life as well as polluting the environment, and form the basis for ensuring the quality of the products to be produced. The guideline VDI/VDE (1998) defines alarms as a state

which requires immediate reaction by an human operator. It further distinguishes between alarms that originate in the technical process and alarms or messages giving information about disturbances in the control system. The latter kind of alarms does not serve the operation of the technical process. Thus, it is not subject of this thesis. The guideline NAMUR (2005) distinguishes between, in analogy with [EEMUA, 2007], among others, *absolute alarms*, *deviation alarms*, and *rate of change alarms*. However, the investigated existing visualization systems (see Section 3.1.2) only support absolute alarms. Hence, this thesis focuses only on this type of alarms.

VDI/VDE (1998) gives recommendations for the ergonomic presentation of messages and reliable perception of signals. From this specification, the configurable parameters shown in Table 3.5 can be deduced. They form the basis for a dedicated Alarm Perspective, as depicted in Figure B.62.

Table 3.5: Characteristic parameters to be considered for the definition of Alarms.

Parameter	Description
Time	Defines the time of occurrence.
Severity	VDI/VDE (1998) recommends to specify different levels of importance for alarms so that an operator is prepared for short-term reactions. By referring to suitable Alarm Behaviors of Data Items (comp. Section 3.2.2), it can be defined which limit values of a Data Item accompany with which severity level. NAMUR (2005) proposes to use the following severity levels: *high-high*, *high*, *low*, and *low-low*. They also were established in practice [Zacher and Wolmering, 2009].
Priority	For presorting alarms by one severity level. It "helps the operator to decide which alarms to deal with when several occur at the same time" [EEMUA, 2007]. particularly if the alarm system gained a significant size. Zacher and Wolmering (2009) propose 15 priority levels. However, small or medium-sized solutions are managed well with less levels.
Description	To inform operators about the cause and place of the particular alarm.
Hysteresis	Prevents alarms to be repeatedly generated if a value varies with a small amplitude around a limit value.
Groups	Reducing the number of messages to a minimum through grouping alarms into logical groups. Only dedicated group messages are presented to the operator.
Notification Class	Another dimension of grouping alarms with the aim of presenting them to human operators. A Notification Class is the smallest unit a dedicated widget can register to.

Alarms must be presented in an appropriate way. Several meta information

about alarms are required to support the operators when reacting on deviations in the technical process. Table 3.6a lists information about an alarm to be presented through a dedicated user interface widget. Additionally, when handling alarms, certain actions on alarm lists must be possible. Table 3.6b presents these actions. However, these aspects belong to the *fixed operative characteristics*, as each user interface must support operators with these tasks on the alarm lists. As a consequence, the transformation has to generate appropriate user interface widgets with suitable interaction means.

Table 3.6: Aspects to be considered when presenting alarms to operators.

(a) Information about alarms.

Attribute	Description
Alarm Time	The time of occurrence.
Acknowledged?	Indicates if the alarm has already been noticed.
Actual Value	The value of the process variable at time of occurrence.

(b) Actions to be performed on alarm lists.

Action	Description
Select	In order to affect several alarms with a single action.
Sort	Different sorting criteria can be used, e.g. time, severity etc.
Suppress	Hide an alarm or a group of alarms because they are not related.
Block	Further actions on alarms are not possible.
Acknowledge	Mark an alarm as noticed.

Figure B.63 distinguishes between an element Local Alarm and an element Remote Alarm. Local alarms are to be presented to local operators through a dedicated user interface widget. Remote alarms enable to inform operators via Email, SMS, telephone, and even via *Social Networks*, such as Twitter or Facebook.

Logging

VDI/VDE (2002) recommends to record operational actions that have been performed manually through the user interface in an operation log. Moreover, technical processes, producing food or drugs, must comply with legal requirements. For instance, the FDA[8] requires to record and archive certain process information for several years, as specified in [CFR 21, 1997].

[8] U.S. Food and Drug Administration

To minimize the amount of data to be stored, a Data Item can be configured to record and archive its values. For this purpose, it provides the attribute Logging (see Figure B.60). It is only a boolean decision, as the method of logging is captured by the transformation as *fixed operative characteristics*.

3.2.3 Presentation Model

Section 3.1.3 concludes with a clear definition of the *Presentation Domain*: User interfaces to monitor and operate technical processes are composed of mimics that represent the physical topology of the plant and show the actual process state. In addition, these user interfaces contain interaction means to influence the technical process and they provide other components that ensure safe operations. Existing visualization systems provide a comprehensive set of user interface widgets, each containing a broad range of configurable parameters. In general, this range of operative characteristics is underpinned by the specifications [VDI/VDE, 1997, 1998, 1999], and more recently by [VDI/VDE, 2011], giving recommendations and regulations for the design of industrial user interfaces.

The Presentation Model aims at a systematic consideration of the aforementioned aspects resulting in a metamodel capturing the operative characteristics of industrial user interfaces. Figure B.21 shows that a user interface is composed of one or more Panels, each representing an individual view on the process. Each Panel is composed of UI Components. They are either elementary or complex (see Section 3.1.3) and inherit the abstract type Type, defined in [OMG, 2010, p. 27, Figure 7.5]. This is due to the fact that Section 3.2.1 introduces the Read Presentation Model Action to establish the interface "Influencing appearance of user interface widgets" (see Figure 3.7b) between Algorithm Model and Presentation Model by sending signals to user interface widgets. As the UML Output Pin is a Typed Element (see [OMG, 2010, p. 228, Figure 11.3]), the "Result" of a Read Presentation Model Action is always a single UI Component, which is the recipient of the particular signal.

Definition 3.1: *A* UI Component *is a coherent presentation unit, which is very similar to the definition of the presentation units introduced in [Botterweck, 2007]. It will always be presented as envisaged by the modeler.*

In the following, it will be described how the Presentation Model captures the operative characteristics of the target subdomain. The main idea is to configure UI Components using three different kinds of properties. They will be explained in detail in Section "Widget Properties". Elementary and Complex UI Components (Section "Elementary UI Components" and Section "Complex Components") inherit

these properties. In addition, they define specific ones. Finally, Section "Navigation" explains the configuration of navigation structures through mimic hierarchies, as suggested by VDI/VDE (1999).

General Characteristics

Considerations about the following characteristics originated from experiences received while developing visualization solutions with dedicated tools:

Cloned Panel/Cloned Component. VDI/VDE (1999, p. 10, Fig. 3) recommends a specific division of the screen to be used to monitor and operate technical processes. For instance, it defines an alarm row on top of each screen. The actual mimic is located in a working area in the center of the screen. As this basic structure applies to all Panels, a template mechanism would lead to a more efficient modeling process. The reference Cloned Panel (comp. Figure B.21) was introduced to indicate that a particular Panel additionally obtains all UI Components of the cloned panel. Similarly, the reference Cloned Component, as depicted in Figure B.21, enables UI Components to inherit all properties from the cloned component that have not already been defined in the cloning component, thus leaving room for individual attributes. This fosters, among others, the consistency of presentations, as required by VDI/VDE (2005).

Multi Lingual Text Definition. In the course of globalization and worldwide communication networks, operators from different countries might get access to a process visualization: Consulting experts from other company locations could be one example. According to VDI/VDE (2005), "the presentation of the information must allow interpretation beyond doubt by following simple rules." Hence, presenting information in the native language of the country fosters to meet this design goal. Consequently, the Presentation Model supports multilingualism through the element Multi Lingual Text Definition (comp. Figure B.23). A Text Item is a single phrase of a particular language. It will be combined together with all translations of the same phrase in a Text Definition. An elementary user interface widget, which contains text such as a Button, only refers to the particular globally defined Text Definition element. An additional advantage of this method is that text definitions can also be reused in other user interface widgets. This fosters, among others, maintainability and sustainability.

Color Definition. As VDI/VDE (2005) requires to ensure consistency, a global Color Definition was inserted into the Presentation Model (comp. Figure B.25). By this means, colors can be globally defined and augmented with descriptive titles such as "warning" or "error". UI components using colors only refer to these titles for the sake of reuse and maintainability.

Image Bundle. Visualization solutions make heavy use of images. However, many images are used in various UI Components. For example, a visualization solution defines several tanks, but each tank defines the same background image. To foster reuse and thus maintainability, a global image library — the Image Bundle (see Figure B.24) — is provided by the Presentation Model. It distinguishes between Fixed Images and Camera Images. Similar to the Color Definition, images can be globally defined for the sake of reuse and augmented with descriptive titles.

Widget Properties

Section 3.1.3 lists numerous parameters, provided by existing visualization systems, that can be used to configure UI Components. The basic idea is to classify these properties, according to their kind of influence on the UI Component, into the categories *Representation*, *Animation*, and *Interaction*. Figure 3.11 shows the main relationship between these classes.

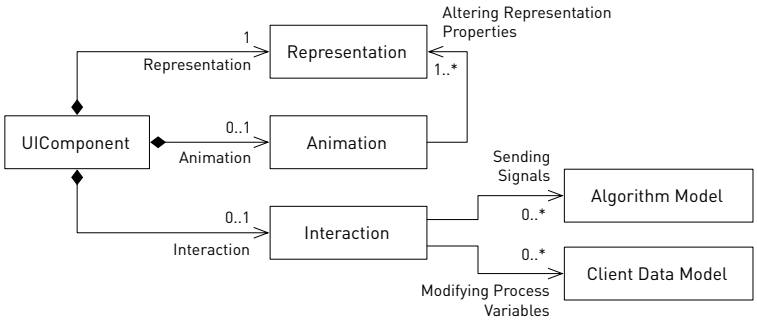


Figure 3.11: Relationship between a UI Component and its configurable parameters, classified by the categories *Representation*, *Animation*, and *Interaction*.

Representation properties define the initial appearance of UI Components. *Animation* properties always refer to representation properties to alter them according to actual data. *Interaction* properties are used to specify how an operator can interact with a particular UI Component and in which effects this interaction results.

Representation Properties. The following three types of *Representation Properties* can be distinguished:

properties being mandatory for all UI Components: These properties, such as Position or Size, were identified in Table 3.1 and realized by appropriate Representation Records as depicted in Figure B.26. Both Figure B.27 and Figure B.28 show these properties in relationship to the element Representation.

properties being relevant only for some UI Components: Figure B.29 presents two more concrete representation types being only applicable either to widgets containing text or to widgets containing an image. The Text Containing Widget Representation refers to a Text Definition (comp. Section 3.2.3) in order to equip the UI Component with a text item while considering multilingualism. Similarly, the Image Containing Widget Representation equip suitable UI Components with images. As it only refers to an Image Resource that is defined within an Image Bundle, also Camera Images can be included transparently.

properties being exclusively used by a single UI Component: Each UI Component defines configurable properties of its own to set individual characteristics of the particular user interface components. For instance, a Combo Box (comp. Figure 3.5e) provides, besides the aforementioned common representation properties, means for configuring a list of items. Discussing these parameters in detail is subject to Section 3.2.3.

Figure B.27 shows that the Representation element is declared to be an abstract element. This forces all UI Component — either elementary or complex — to derive its own representation element from the abstract ones. In this way, a high level of reuse as well as individuality is ensured.

Animation Properties. UI Components are the smallest building blocks to represent the operative states of technical processes through graphical displays. For this purpose, each UI Component can contain several *Animation Properties*, each influencing one or more representation properties. Animation properties follow the principle of defining *what* representation property should be altered, *when*, and under *which* conditions:

what: An animation property can either be a Continuous Animation Record to display analog values or a Discrete Animation Record to display discrete states (comp. Figure B.32), as distinguished by VDI/VDE (1999). Continuous Animation Records directly influence representation properties by means of actual data. As a result, a UI Component can be configured to

represent a fuel level of a tank by connecting its Height property to a process variable using a continuous animation. Discrete Animation Records predefine a set of representation properties and activate one of them if a certain condition is met. Various states of UI Components can be predefined and established in this way. Table 3.7 gives an overview of available discrete animation properties, lists further kinds of animation properties, and their exemplary usage.

when: A UI Component changes its appearance if a process value or a state in the Algorithm Model changed. For this purpose, animation properties establish both interfaces "Impacting on user interface widgets" and "Influencing appearance of user interface widgets" depicted in Figure 3.7b by referring to Signals (comp. Figure B.32). Data Items of the Client Data Model are linked to UI Components of the Presentation Model by this means. Additionally, results of custom functionality expressed in the Algorithm Model can be presented through the user interface: An *Animation Property* might be triggered using a Send Signal Action. It evaluates the data carried by an attribute of the Signal and alters the representation of its widget accordingly.

which: Discrete Animation Records replace a currently active representation property with one of a set of predefined properties if a certain condition is met. These conditions can be configured using the Comparator (comp. Figure B.32). By this means, a UI Component can be configured to change its background color from yellow to red if a process value exceeds a defined reference value. A Range Comparator enables to define a value range in which a condition is met.

Table 3.7: Different kinds of animation properties for a precise specification a UI Components behavior.

Animation Property	Description
Discrete Animation Record	
Representation Animation	Defines arbitrary Representation Records (comp. Figure B.26) and activates them under certain conditions. (A change of background color provides information about the operating state of a field device.)
Image Animation	Components containing images can define a set of images that are activated if a certain condition is met. (A change of an image indicates the status of a pipe.)

Animation Property	Description
Text Animation	Components containing text can define a set of texts that are activated if a certain condition occurs. (A change of text allows to capture the status of field devices, e.g. "ON" or "OFF".)
Further kinds of animation records (comp. Figure B.34)	
Value Output Animation	The value of a process variable will be displayed in a way that is specific to the particular UI Component. If it is a UI Component containing text, the output might be a string or a numerical value. A Value Output Animation can define a Pre Text and/or a Post Text to add a meaning to the presented data, thus making it definite as required by VDI/VDE (2005).
Indicator Animation	This animation property is only applicable to UI Components defining an Indicator (see Section 3.2.3).

Figure B.35 depicts the topmost Animation element, providing standard animation properties being available in each user interface widget.

Interaction Properties. Modeling a UI Component to react on a particular interaction of an human operator means to specify *how* the operator can interact with the particular UI Component and *what* action is caused as a result. As Figure B.36 shows, this principle is realized by a relationship between suitable Trigger elements and pertinent Interaction Effect elements — a trigger, as a way to input information, causes an effect as method to process this data. The means for information input and, thus, the triggers strongly depend on the capabilities of the hardware platform. Requirement 1 on page 22 determines the modeling language to reside at CUI level using basic haptic input techniques: Basic off-the-shelf devices (workstations, mobile devices), but also automation specific platforms, such as *Industrial Panel PCs* for close-to-machine interaction, are used to monitor and operate technical processes. In this context, VDI/VDE (2002) distinguishes between various sorts of *keyboards* and different kinds of *pointing devices*. For instance, while workstation PCs (platform: Figure 1.2c) are equipped with a physical keyboard and a mouse, smart phones (platform: Figure 1.2b) contain virtual keyboards and touch screens. Industrial panel PCs (platform: Figure 1.2a) are equipped with function keyboards and arrow keys or track balls. To take these interaction types into consideration, suitable Triggers were introduced. They are shown in Table 3.8, together with available effects.

Table 3.8: Interaction properties for modeling UI Components which are sensible to interactions of human operators.

Interaction Property	Description
Triggers (comp. Figure B.36)	
Click, Double Click, Touch	These triggers are sensible to basic pointer device interaction, either using a computer mouse or the fingers on a touch screen.
Key Sequence	Using this trigger, a UI Component is capable of reacting to keyboard interaction. With it, it is possible to define arbitrary key sequences also for industrial panel PCs.
Submit	This trigger can be applied to UI Components that are capable of receiving data from an human operator (e.g. to set points). Using this trigger, a UI Component submits the entered data to the configured effect after performing the respective submit gesture (typically by hitting the Enter-key on the keyboard).
Effects (comp. Figure B.38)	
Send Signal	This effect realizes the interface "Triggering functions on user interaction" (see Figure 3.7b) between the Presentation Model and the Algorithm Model causing a state change in the Algorithm Model.
Client Data Model	Realizing the interface "Modifying Process Variables" between Presentation Model and Client Data Model, these types of effects enable configuring UI Components to influence the relationship of the user interface and the process data on user interaction. By this means, reading and writing particular process variables can be triggered explicitly. Furthermore, Subscriptions can be started or stopped.
Navigation	Effects of this type cause a Navigation Flow (comp. Section 3.2.3) to establish a transition from the current Panel to the desired one.
Print	For maintenance or assessment reasons, it might be necessary to archive data sets in paper form. Hence, this effect enables sending the current screen, especially containing large data sets in a table, to a printer.

Elementary UI Components

VDI/VDE (1999) suggests display elements to compose mimics. These are in detail geometrical elements, static images, and elements capable of output texts and

values. Additionally, basic interaction elements are required to write information to the field devices or to change a state in the visualization solution. As Section 3.1.3 illustrates, existing visualization systems provide a comprehensive set of user interface widgets, each containing a broad range of configurable parameters. While Section "Widget Properties" systematically classifies the configurable parameters by means of *Representation*, *Animation*, and *Interaction* properties, this section aims at providing a systematic view on user interface widgets.

For keeping the modeling language clear and simple while simultaneously providing as powerful methods for user interface modeling as existing visualization systems do, the idea is to derive a limited set of elementary user interface components. It forms the basis for building user interfaces of visualization solutions of almost any complexity. Figure B.40 presents Elementary UI Components that were derived from the investigation of Section 3.1.3 and from the recommendations in VDI/VDE (1999).

Definition 3.2: *An* Elementary UI Component *is an atomic building block for composing user interfaces.*

Each Elementary UI Component defines their own specific *Representation*, *Animation*, and *Interaction* elements, which are derived from the superior ones. The reason for this is that each widget is characterized by very specific properties. For this purpose, besides defining own properties being only applicable in the particular widget, each widget inherits properties that are shared with other widgets. Fostering precise modeling and, thus, reducing the need for verification rules are the aims of this approach. Table 3.9 explains the discussed specific properties of the provided Elementary UI Components.

Table 3.9: Elementary UI Components with their specific characteristics.

UI Component	Description
Geometrical Objects	
Ellipse	Figure B.46 shows that Ellipses are defined by using a Center Point, a X Radius, and a Y Radius.
Polyline	A line requires the configuration of at least two Points, as formalized in Figure B.50.
Polygon	Drawing arbitrary shapes requires the definition of at least three Points (comp. Figure B.51).
Widgets	

UI Component	Description
Alarm Control	As these types of components must always provide a standard feature set — searching, sorting, and acknowledging alarms —, configurable parameters are limited to the definition of which alarms should be presented by this component (see Figure B.42). For this purpose, the *Animation* refers to alarms that were embraced in different Notification Classes (comp. Section 3.2.2).
Button	As Figure B.43 illustrates, this widget type might contain both text and images. Hence, it provides suitable animation records, besides the ones that are available per default.
Gauge	Informing about current process values and their respective speed of change requires to configure indicator needles and appropriate scales (comp. Figure B.47).
Image	This type of widget specifies an Image Resource and provides the suitable discrete animation property (see Figure B.48).
Text Label	These types of widgets are used to output process values numerically and discrete, respectively. Hence, the desired text definition is required initially. Animation records are provided to alter this text (comp. Figure B.56).
Trend	Observing process values over a period of time requires to configure this period using suitable scales and trend graph indicators (see Figure B.59).
Value Input Widgets	
Check Box	Enables configuration of widgets providing a check mark to be set and a text label next to it (see Figure B.44).
Check Box Array	Combines arbitrary Check Box widgets to provide a list of options from which an arbitrary number can be chosen. Figure B.44 illustrates the relationship between both types of components.
Drop Down	Defines a set of options in form of a list from which one or more options can be chosen (comp. Figure B.45). Particularly, this widget allows to modify the list of options using a *Value Output Animation*.
Input	Realizes multi and single line input fields to set numerical or text values to be written to the process (comp. Figure B.49).
Radio Button	Enables configuration of widgets providing a round check mark to be set and a text label next to it (see Figure B.52).
Radio Button Group	Combines arbitrary Radio Button widgets to provide a list of options from which only one option can be chosen. Figure B.52 illustrates the relationship between both types of components.
Slider	Enables configuration of widgets for submitting a nominal value of a given range. It requires to specify a scale and an indicator that is also used as its handle (comp. Figure B.53).

Complex Components

Section 3.2.3 identifies and specifies Elementary UI Components as basic building blocks to model user interfaces for monitoring and operating technical processes. Aggregating these elementary building blocks in any complex components primarily aims at reducing development efforts. The specification VDI/VDE (1999) explicitly recommends to provide predefined display elements, e.g. to make these complex components available through dedicated libraries, which would foster reuse. Also Völter (2009) proposes to provide libraries, containing instances of the language model, as this will also reduce complexity of the language. Moreover, complex components support modelers through giving them a certain already predefined layout.

Definition 3.3: *A Complex UI Component is a component that is completely composed of elementary elements or other complex components, but it expects them to be shaped by a particular structure.*

To adhere to Definition 3.1 on page 51, a complex component constitutes a new coordinate system for its aggregated widgets. Basically, three types of complex components are provided in the language model, as Table 3.10 shows.

Table 3.10: Complex UI Components.

UI Component	Description
Simple Container	Embraces UI Components without constraining a certain layout. It only defines a new coordinate system for the containing components. Figure B.41 introduces this component together with its property set.
Table	Figure B.54 and Figure B.55 show that each table cell can contain one UI Component. If a Table Row is configured as Template, it is receptive for data structures, received through a *Value Output Animation*, that accompany with the structure of the table row. By this means, the table can be used to display large data sets, such as operation protocols.
Tree	Displaying hierarchies is particularly important for building structured navigation means, which represent the actual mimic hierarchy. Figure B.57 and Figure B.58 show how these tree-like structures can be created.

To store complex components into libraries for reuse in other projects, an investigation on required interfaces is necessary. Figure 3.12 depicts the required information that are defined outside of a particular UI Component. Consequently,

components of a library cannot be a valid, which can be tolerated as long as this will be fixed in the future — when a model will be created. This means that either the model must be tailored to provide the required interfaces or the complex component must be adopted.

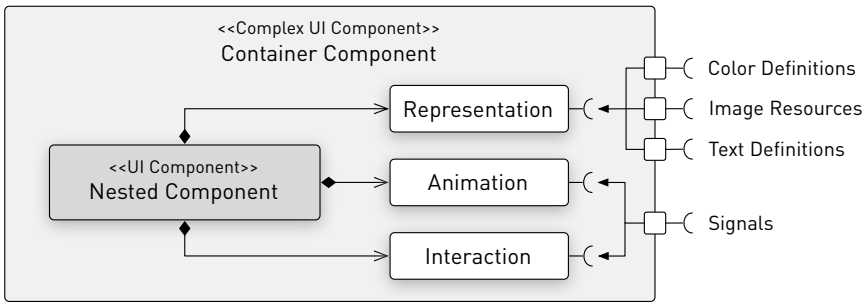


Figure 3.12: Complex UI Component and its properties being defined outside of the component itself.

Navigation

A single mimic only represents a part of the plant or the technical process with a certain level of detail. Navigating through a hierarchy of mimics is technically accompanied by calling up a mimic from a particular display through an appropriate interaction property (comp. Section "Widget Properties"). In this respect, it is important to consider the following facts:

(1) It can be specified which mimics are allowed to call up from a particular display.

(2) It must be possible to return to a previously selected display. Thereby, both forward and backward paths do not necessarily have to be the same.

Figure 3.13 presents an example navigation with four displays. Navigating from "Display 1" to "Display 3" must be done stepwise, the return path is arbitrary. "Display 2.1" can only be blended in on top of "Display 2" (e.g. to show additional details of a particular field device). This is stressed by the dashed arrow.

Ensuring that each display can be reached and that there is always at least one possible return path can be achieved by making the navigation paths explicit in the model. By this means, Panels are required to be connected via Navigation Flows, as Figure B.22 depicts, which results in an explicit *navigation perspective* with the possibility of verifying the reachability of Panels (comp. Section 5.2). A Navigation Flow can change the actual display or it can open a new display

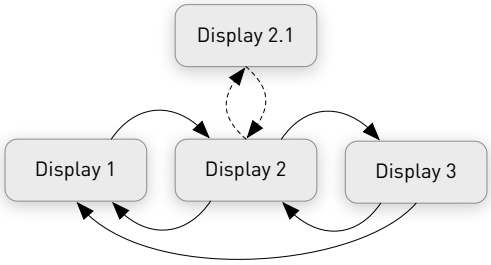


Figure 3.13: Navigation path through the displays of a visualization solution. The dashed arrow indicates that "Display 2.1" will be blended in on top of "Display 2".

(opening a new display on top of another allow to specify dimensions of the new display). A Navigation Flow can be activated as follows:

Navigation Effect: A Navigation Flow refers to a Navigation Effect to change a display based on an user interaction.

Signal: Since a Navigation Flow is derived from Type, it is receptive to signals. When using the reference Trigger, which is depicted in Figure B.22, it can be precisely specified which Signals cause the Navigation Flow to change a display. By this means, an overlay display can be opened on changing the value of a particular process variable. Sending Signals from the Algorithm Model enables the definition of far more complex and powerful navigation means.

Concluding, the navigation subsystem can be seen as state machine treating the Panels as states, Navigation Flows as transitions, and Navigation Effects as well as Signals as triggers.

3.3 Language Model Constraints

Not every fact about the target domain can directly be expressed in the *Core Language Model* with the required precision and clearness, as required for automatically processing resulting models. *Language Model Constraints* are means for including additional invariants into the language model. They will be defined on modeling elements or on relationships between them [Strembeck and Zdun, 2009] if the wellformedness of these modeling entities depends on a particular configuration of one or more other elements or relationships. Basically, three types of these constraints can be distinguished in the context of the core language model, which was deduced in Section 3.2:

Intra-Submodel Constraints: Facts about the target domain, which are only relevant within a particular submodel, can be considered using this type of constraints. Hence, intra-submodels constraints are only defined on elements and relationships between them within a certain submodel.

Inter-Submodel Constraints: This kind of invariants are defined on relationships between elements belonging to different submodels. Consequently, inter-submodel constraints consider target domain facts about the interfaces between submodels, which have been identified in Figure 3.7b.

Model-Integrity Checks: This type of invariants do not add additional static semantics to the language model. More importantly, they add an additional layer atop the core language model that enables to identify modeling errors at an early stage (comp. Requirement 3 on page 25).

The following identifies and explains selected constraints on the core language model according to their belonging to one of these aforementioned types of constraints. Their necessity will be discussed on example configurations using UML object diagrams.

3.3.1 Intra-Submodel Constraints

Constraints of this type ensure certain facts about the target domain that only impact an individual submodel. Hence, each submodel will be treated individually hereafter.

Algorithm Model

Basically, as the Algorithm Model is an Executable UML realization, the constraints defined in the UML specification [OMG, 2010] and the FUML specification [OMG, 2008] were adopted. However, the language model constraints of both specifications leave room for interpretation. Hence, additional constraints were introduced for the sake of preciseness. For instance, Read Link Actions (comp. Figure B.19 and [OMG, 2010, p. 275]) enable navigating along an association to gain access to the connected objects. The configuration of these actions is not trivial, as Figure 3.14 illustrates: The Input Pin "P1" determines the navigation's initial point, the elements "LE1" and "LE2" of the type Link End Data refer to the association ends "AE1" and "AE2". UML does not formally constrain that these ends must belong to the same association (in this example it applies to configuration "A1"). Also, it does not specify that the Association End object "AE1" must be contained by the Class "C1", which must be addressed by the involved Input Pin "P1".

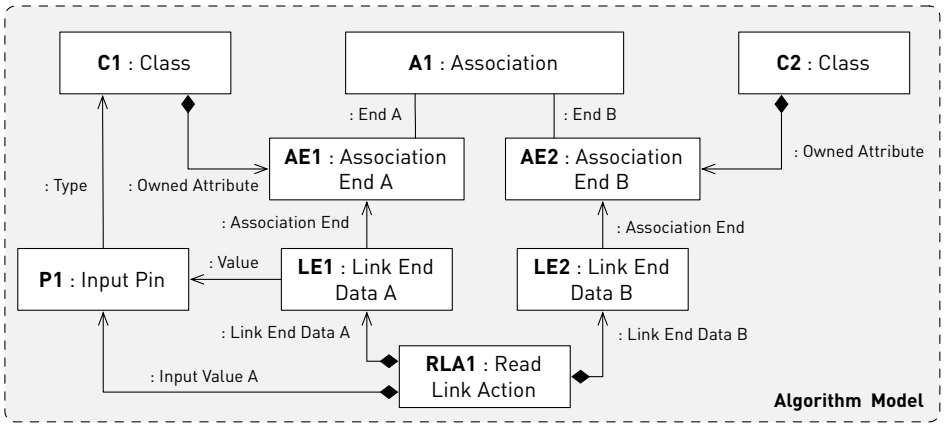


Figure 3.14: Valid example configuration of a Read Link Action.

Hence, appropriate constraints are introduced to the core language model, particularly to the Algorithm Model, in order to provide a more comprehensive degree of preciseness and less ambiguity. It enables modelers to create more precise models. Consequently, it facilitate to elaborate on less complex transformations, but it does not forestall compatibility to the UML specification.

Client Data Model

Most crucial in the Client Data Model is the correct information mapping between the Logical Data Perspective and the Technical Data Perspective. Although the transformation is responsible for this mapping (comp. "Information Mapping" in Section 3.2.2), language model constraints are intended to ensure the wellformedness of models. Figure 3.15 shows a wellformed Client Data Model configuration. Both Data Items, defined in the Logical Data Perspective, point to Server Data Items of the Technical Data Perspective. In this particular case, an OPC XML-DA server with appropriate items was defined. Constraints ensure that both types can be mapped: "SDI1" of type "boolean" can be mapped to "DI1" of type "Boolean". To map "SDI2" of type "ArrayOfBoolean" to "DI2" an appropriate Structured Type was configured.

To add another data server specification, constraints are necessary to ensure the correct mapping between the data types of the new data server specification and the data types provided by the core language model, as depicted in Figure B.5. Although the concept of the Generic Server was introduced in "Technical Data Perspective" of Section 3.2.2 to reduce efforts when integrating new data server specifications, this concept requires additional constraints. Figure 3.16 shows

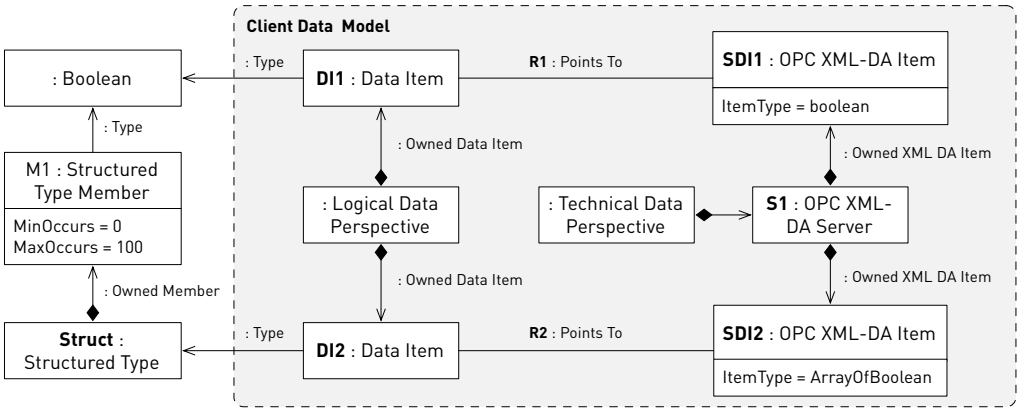


Figure 3.15: Valid example configuration of the Client Data Model: constraints ensure a correct information mapping between the logical and technical data perspective.

a wellformed example configuration: It depicts two different Generic Servers (indicated by the Type parameter), each containing Generic Items. It is ensured through appropriate constraints that the Generic Items of one particular Generic Server feature the same set of Generic Parameters (characterized by their property Name). In Figure 3.16, the Generic Server "GS1" defines items, each is characterized by an "Address". "GS2" contains items featuring "AccessPoint" properties. Additionally, it must be ensured that the concrete Generic Server instances can be mapped to Data Items.

Presentation Model

Although the Presentation Model is precisely formalized — particularly with respect to forcing UI Components to specify their own concrete Representation elements (comp. Section 3.2.3) —, appropriate language model constraints provide less ambiguity and, thus, facilitate models of visualization solutions to be correct and, more importantly, complete. Basically, it must be ensured that UI Components are of the same type if they are connected with the relationship Cloned Component (comp. Figure B.21). Figure 3.17 contrasts a correct and an incorrect configuration.

3.3.2 Inter-Submodel Constraints

As stated above, constraints of this type capture facts of the target domain with respect to the interfaces between the deduced submodels. For instance, when

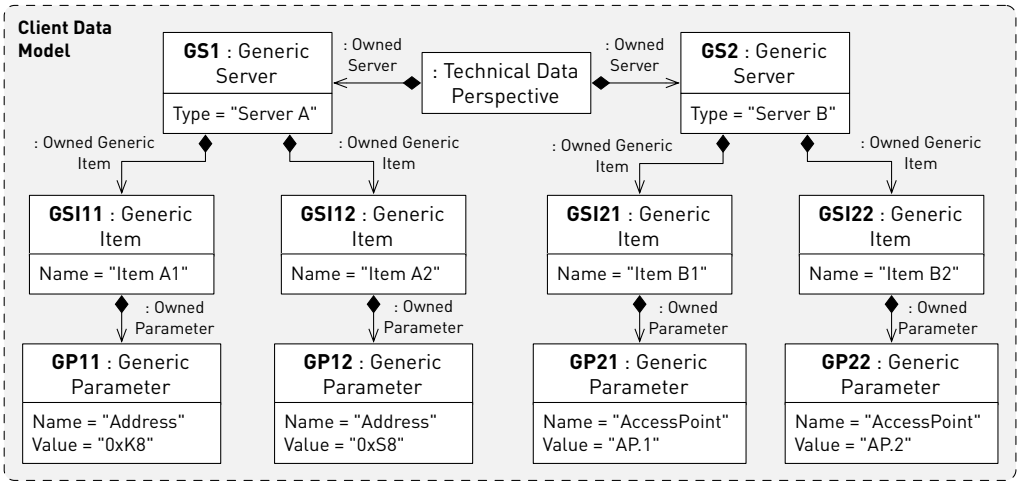
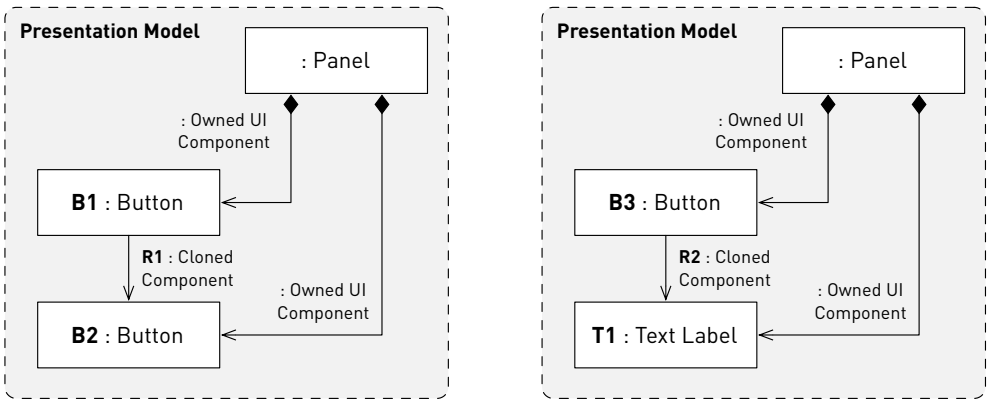


Figure 3.16: Valid example configuration of the Client Data Model; language constraints ensure the correct usage of the Generic Server elements.



(a) Correct Configuration.

(b) Incorrect Configuration.

Figure 3.17: Correct (a) and incorrect (b) configuration of a Cloned Component relationship between two UI Components.

configuring Data Items to be written using dedicated actions of the Algorithm Model and, thus, serving the interface "Modifying Process Variables" (comp. Figure 3.7b), four modeling entities need to be involved, as Figure 3.18 illustrates. The Write Data Item Action contains an Input Pin, representing the value to be written to the Data Item. Both the Input Pin and the Data Item must be of the same type. Hence, in order to ensure wellformedness of the element "E1", the relationship "R3" must refer to the same object as relationship "R4". These constraints cannot be statically expressed using the means provided by the *Core Language Model*. Thus the definition of appropriate constraints is required.

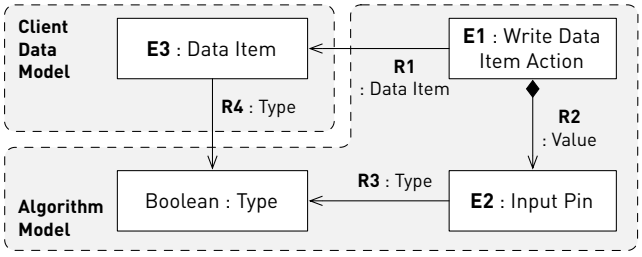


Figure 3.18: Illustrating the need for additional language model constraints: Both relationships "R3" and "R4" must refer to the same object in order to ensure wellformedness of the relationship "R1".

More complex invariants are necessary in the course of establishing the interface "Impacting on user interface widgets" between Client Data Model and Presentation Model (comp. Figure 3.7b). Particularly, Discrete Animation Records, as depicted in Figure B.32, contain Comparators. A concrete comparator type (comp. Figure B.33), however, depends on the connected Data Item type. Since the Data Item and the Animation Record are indirectly interconnected, because they refer to the same Signal, as depicted in Figure 3.19, only the definition of additional static semantics allow to ensure wellformedness. In this particular case, it has to be assured that an element Boolean Comparison is used, because the Data Item, connected through the Signal "S1", refers to the type "Boolean".

After all, the wellformedness of every Animation Record depends on the data type being defined in Data Items. In this context, a Continuous Animation Record requires data items to be configured with Numeric Types. The wellformedness of Value Output Animation Records additionally depends on the UI Component to which they have been connected: (1) Being connected to a Button, Input field, or Text Label, this type of animation requires data items to be configured with any Primitive Type. (2) Being, for example, connected to a

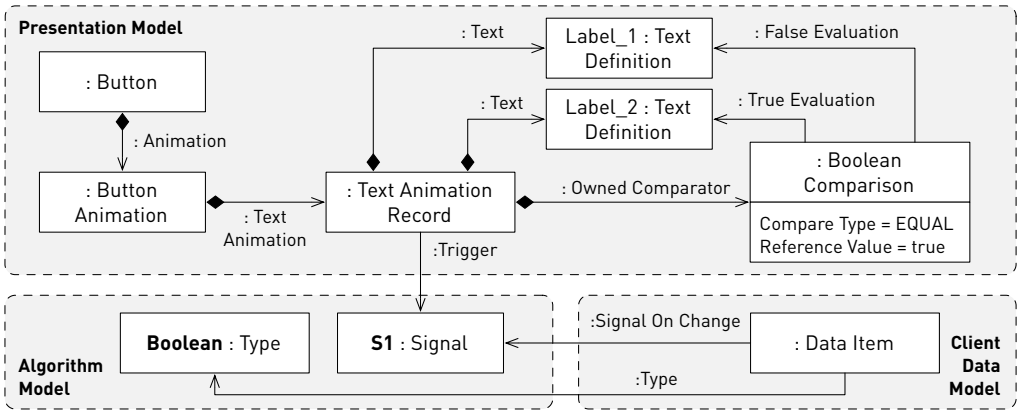


Figure 3.19: Illustrating the need for language model constraints: Concrete Comparator types depend on the type of the connected Data Item.

Check Box requires data items configured with a Boolean Type. In case of a Check Box Array, a Structured Type with boolean type members is required.

Concluding, *inter-submodel constraints* are mainly used to ensure type safety between Data Items and UI Components of the presentation model or Actions of the algorithm model.

3.3.3 Model-Integrity Checks

The more models grow the more difficult it gets for modelers to maintain integrity of these models while expressing their operative characteristics with it. For instance, when defining navigation paths along several Panels (comp. Section 3.2.3), a Navigation Flow might refer to a Navigation Effect that is not defined in the Panel from which the Navigation Flow originates. Transforming such model will result in correct source code and, thus, in a program that can be executed. However, an important view on the process might not be reachable through human operators during runtime — which might not be recognized until an hazardous situation occurs.

Model-Integrity Checks are specific constraints that add no additional static semantics to the core language model. Consequently, the transformation will create executable source code, even though these constraints are violated. Meeting these constraints, however, might increase the correctness, robustness, and reliability of the generated solution during runtime and ensures its desired behavior. In the following, a selection of provided model-integrity check constraints will be discussed:

(1) If a Navigation Flow refers to Navigation Effects to establish the transi-

tion from one Panel to another one through a user interaction, the Panel from which the Navigation Flow originates must define the required Navigation Effects. Otherwise, a Panel must only contain Navigation Effects which subsequent Navigation Flows refer to. In this way, it is ensured that there are no *blind* interaction elements on the screens.

(2) UI Components can be configured so that human operators can interact with them using Key Sequences. To prevent conflicts during runtime, it must be ensured that a particular Key Sequence is only defined once per Panel.

(3) Classes of the Algorithm Model require Receptions to be configured to be able to receive a particular Signal (comp. Figure B.3). To assure that each Transition of a contained State Machine can fire and that Accept Event Actions can be served, it must be ensured that a Reception was defined for each Signal to be received. This reduces the likelihood of occurrence of situations where the program execution is blocked.

(4) It must also be ensured that the recipient of a Send Signal Action defines a Reception for the particular Signal. If no appropriate recipient is defined, the signal might either be ineffective or unnecessary. Both cases require further investigation by the modeler.

(5) To connect Data Items with UI Components (to reflect the actual process states), both of these Data Items and the UI Components refer to a Signal. If a UI Component defines an Animation Record that is connected with a Signal, only one Data Item is allowed to refer to this particular signal. Otherwise, it cannot correctly be determined which Data Item causes a certain appearance of the UI Component.

The set of model-integrity check constraints is open for expansion, so that future requirements can be met. For instance, supporting new data server specifications using the Generic Server concept probably requires new language constraints. These can also be added to consider usability or even safety measures. In conclusion, it has to be mentioned that it would be naive to believe that every issue, which can occur at runtime, can be detected through model-integrity checks. They should rather be seen as a sound basis for correct runtime behavior, because many issues can already be solved before transforming the model into source code.

3.4 Language Behavior Definition

"The [behavioral] semantics of a software language describe what happens in a computer when a [model] of that language is executed" [Kleppe, 2009]. Based

on this statement, a *translational semantics* description ought to explain how the particular data, which is captured in an individual model as instances of the elements of the core language model, will be translated to suitable runtime artifacts. As a result of Section 2.4, formulating these mappings with the means of transformations is an efficient method to express behavioral semantics in terms of model-driven software development. However, although transformations are appropriate means to formulate semantics, they are inappropriate measures for communicating the semantics of modeling languages.

Consequently, this section aims at explaining the behavioral semantics of the domain specific modeling language to be developed in a less formal and, thus, more descriptive fashion. Fundamental relationships of the modeling language are exemplified using UML object diagrams for expressing instances of model elements and UML activities to discuss their runtime equivalents. In this case, when a visualization solution in its simplest form is in use at runtime, it has to find answers to the following questions:

(1) Process Communication: How can it be managed to continuously feed the local *data pool* with current process data?
(2) Alarm Management: How can it be ensured that operators are appropriately informed about exceptions and deviations in the process?
(3) Specific Requirements: How can very specific requirements be taken into account, e.g. if a bit sequence returned by a sensor needs a specific interpretation?
(4) Reflecting Process States: How can process data be visually prepared so that operators get an appropriate understanding of actual process states.

Against this particular background, Figure 3.20 depicts the main activity within a visualization solution at runtime. It additionally shows which runtime aspects emerge from the elements of the particular submodel (comp. Section 3.2). Using a *Subscription* that was configured in the Client Data Model, the communication part of the visualization solution manages that the local data pool is always filled with the most recent process data. The change of a data item value leads to both checking if an alarm condition is violated and throwing of the configured event. As the data value needs a specific interpretation, a Boundary was defined in the Algorithm Model. Now, at runtime, the particular event causes the execution of the manually written code. In turn, this throws another event causing a particular *Animation Property* to be evaluated, which is defined in the Presentation Model.

In the following, the characteristics of each partition of the activity depicted in Figure 3.20 are individually explained. Taken this investigation together, it forms the basis for deducing transformations in Section 5.3.

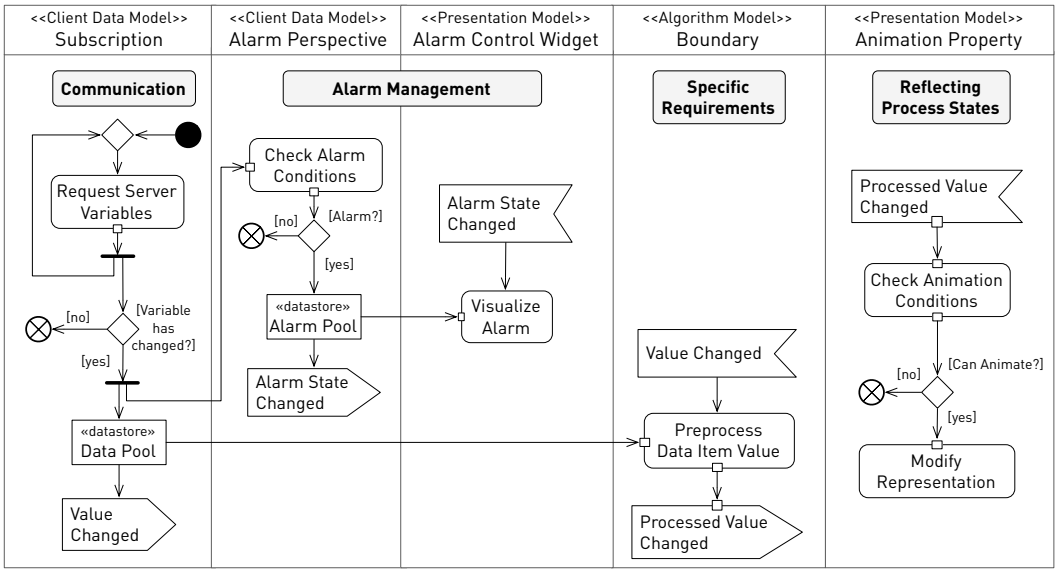


Figure 3.20: Demonstration of the general language behavior by means of a simple visualization solution at runtime. It can be seen which runtime aspects emerge from the elements of the respective submodel, deduced in Section 3.2.

3.4.1 Process Communication

To keep a local data pool always up-to-date, the core language model provides the modeling entity Subscription, as depicted in Figure B.61. A Subscription contains Monitored Items, which in turn refer to Data Items. Thereby, a Monitored Item captures additional information that is required for automatically observing process variables. Figure 3.21 presents an example configuration with two Server objects that were defined within the Technical Data Perspective. Each provides one process variable. The Logical Data Perspective defines appropriate Data Items and groups them into a Subscription. A particular characteristic is that different data server specifications are combined within a single subscription.

As Figure 3.21 depicts, subscriptions can be configured to manage process variables of different concrete servers. A subscription's particular counterpart at runtime, depicted in Figure 3.22, however, makes use of the most suitable characteristics of an individual data server specification. In this way, Figure 3.22 shows that the single subscription of Figure 3.21 resulted in (1) an OPC XML-DA equivalent (comp. [Lange, Iwanitz, and Burke, 2010, p. 60] and [OPC Foundation, 2004]) that is an asynchronous data transfer method. It allows a reduction of the network load because, only changed values are transmitted. Additionally, it

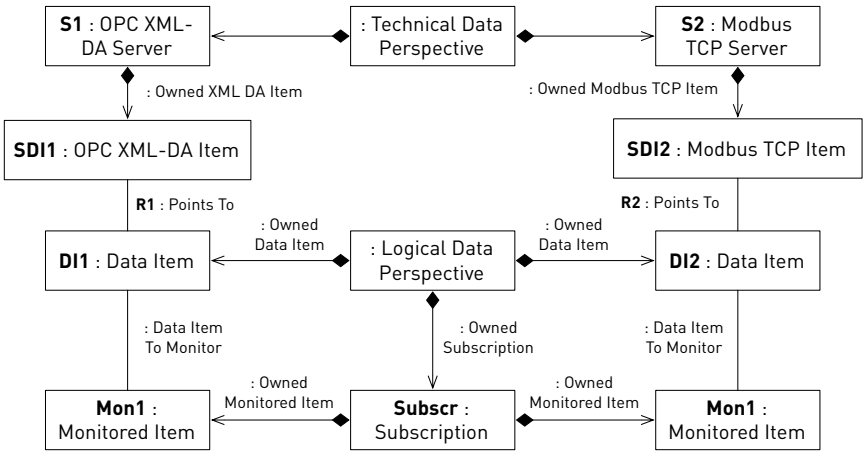


Figure 3.21: Client Data Model configuration that ensures the runtime solution always feeding a local data pool with current process values. A special characteristic is that a Subscription can be defined with data items of different data server specifications.

resulted in (2) a fall-back realization for Modbus TCP using a synchronous polling mechanism. Nevertheless, due to the grouping under a single subscription in the model, the runtime subscriptions depend on each other to some extend: They will be started at the same time. Applying these principles consequently also throughout other parts of the language can lead to the best possible performance at runtime.

3.4.2 Alarm Management

To ensure that operators are kept informed about deviations and exceptions in the technical process to be monitored and operated, Data Items allow to configure certain bounds using the Alarm Behaviors, as depicted in Figure B.60. For instance, Figure 3.23 shows that the Data Item "DI1" defines four specific limit values. Only the Alarm "A1" gives the specific limit values a particular meaning by indicating whether it is an upper or a lower bound.

When checking the alarm conditions of a process variable at runtime, the defined Limit Values determine which Alarm Behavior becomes active. Thus, they determine a certain mutual exclusive state of the alarm (comp. Figure 3.24a) in analogy with the severity levels, as introduced in Table 3.5[9]. Each alarm state in turn is characterized by a state machine, in analogy with [EEMUA, 2007, p. 111,

[9] According to the following mapping: *HH – High-High*, *H – High*, *L – Low*, *LL – Low-Low*.

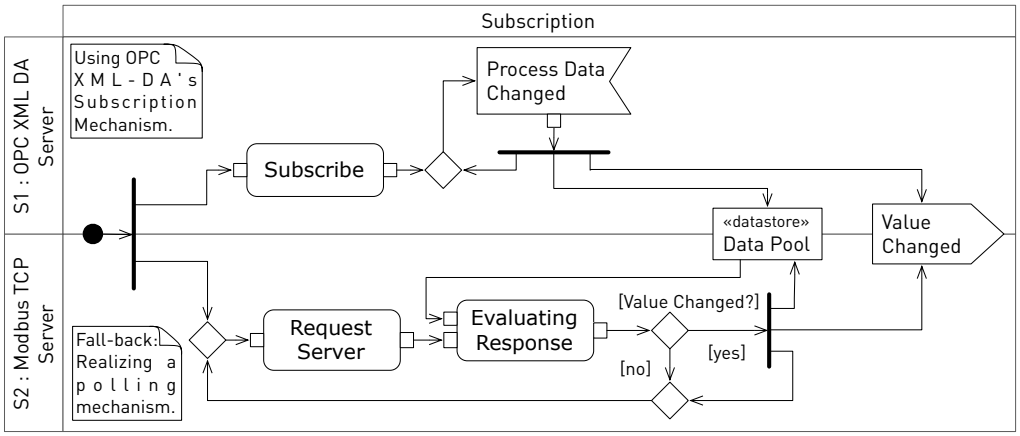


Figure 3.22: Runtime equivalent of the configuration depicted in Figure 3.21: Although a single subscription was modeled, it results in an appropriate communication stub per data server during runtime.

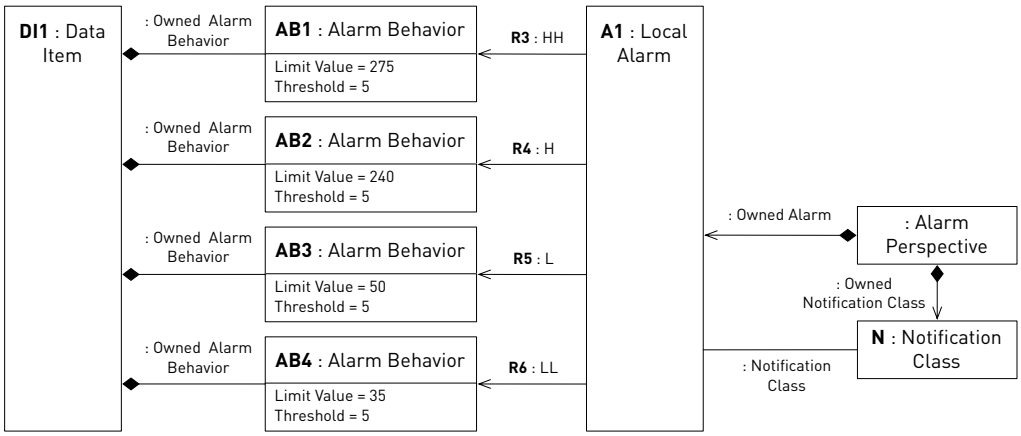
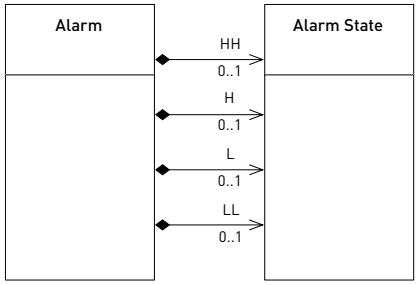
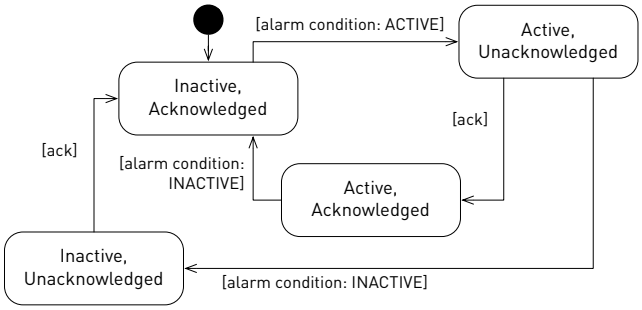


Figure 3.23: The Data Item "DI1", configured in Figure 3.21, defines four specific limit values; an Alarm gives them a certain meaning by stating whether it is an upper or a lower bound alarm.



(a) At runtime, an Alarm Behavior is represented by an Alarm State.



(b) State machine of a single Alarm State.

Figure 3.24: Runtime behavior of an Alarm element: Alarm Behaviors are represented as mutual exclusive Alarm States; an alarm state in turn can be seen as a state machine.

Figure 20], as depicted in Figure 3.24b. The "Active" states indicate that a situation currently exists which causes a certain severity level. An alarm state can further on either be "Acknowledged" or "Unacknowledged", which indicates if an operator took notice of the alarm or if the issue was solved. Consequently, firing a transition of the state machine, depicted in Figure 3.24b, is caused either by an operator interaction or by an exceptional situation in the process. Hence, these state transitions will be recorded in a log file at runtime, as e.g. required by the specification [NAMUR, 2005].

To present alarms to operators and, thus, to allow them to take notice of these alarms, a dedicated Alarm Control widget has to be configured in the Presentation Model. It features a specific Alarm Control Animation property that allows to refer to existing Notification Class elements (comp. concrete element "N" in Figure 3.23 and Figure B.42). All Alarm elements that were grouped by these elements will be presented in the widget at runtime. Particularly, the widget only presents unacknowledged states of an alarm. Figure 3.25 shows an

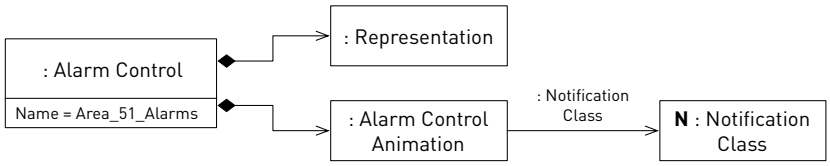


Figure 3.25: Example configuration of an Alarm Control widget. Its characteristic Alarm Control Animation property allows for referring to selected alarms that were grouped by particular Notification Classes. In this example, it only presents the Alarm "A1", as it is connected to this widget through the element "N" (comp. also Figure 3.23).

example configuration.

3.4.3 Specific Requirements

If a process value changes, it needs to be displayed appropriately by the visualization solution. However, these values can also be raw sensor values that need a specific conversion before displaying it. Figure 3.26 shows a configuration of a Boundary element to realize this requirement. It contains two interfaces, one for Signals to be received (Required Interface) and another one for Signals to be sent (Provided Interface). In this example, if a process value changes at runtime, an event will invoke the manually written source code behind that boundary. This is indicated by the connection between Required Interface and Data Item "DI1" through the Signal "Value Changed". Once the manually added routines were executed, possible results will be available at the Provided Interface. Other components that have been registered to the Signal "Processed Value Changed" can use this data as well.

3.4.4 Reflecting Process States and Intervention

If a Text Label component registers to the Signal "Processed Value Changed" (comp. Figure 3.26), it can appropriately display the converted process data value, which will be returned by the Boundary (comp. Section 3.4.3), to operators. Figure 3.27 shows, for this purpose, the required configuration using a Value Output Animation. Due to the relationship between this Animation Record and the Data Item "DI1" through the Signal "Processed Value Changed", the Text Label is intended to write out the converted process value at runtime.

Figure 3.27 also introduces a Text Label Interaction element. It configures the Text Label to allow operator interactions using a *Double Click* with a pointing

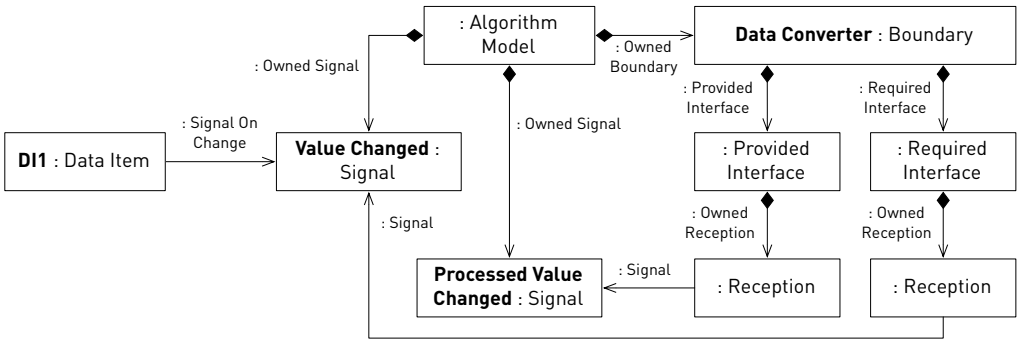


Figure 3.26: Example configuration of a Boundary element intended to integrate application specific code, realizing in this case different methods to convert process data. It expects the data to be converted on the Required Interface and returns the results through the interface Provided Interface.

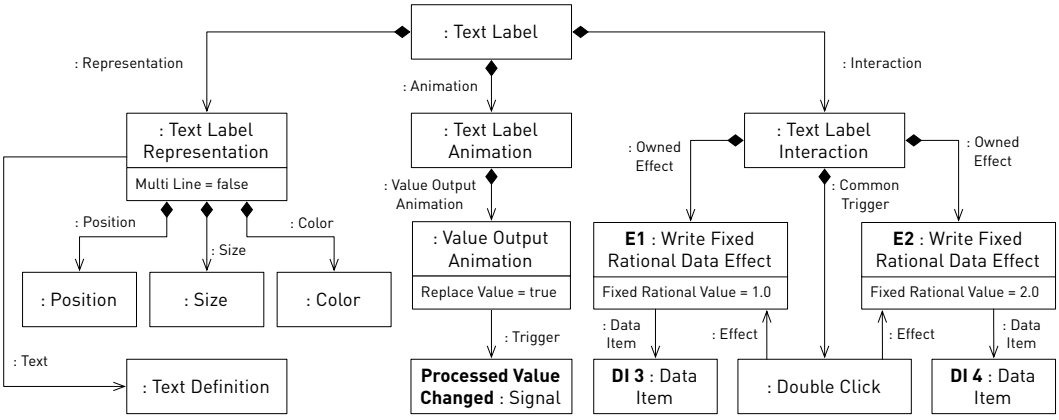


Figure 3.27: Example configuration of a Text Label widget.

device, which is defined by referring to a suitable Trigger element, as illustrated in Figure B.36. This trigger causes two Write Data Item Effects, each in turn writing a particular value to a Data Item, thus illustrating how an atomic write operation can be configured. Assuming that each Data Item belongs to another data provider, Figure 3.28 shows that the particular write operations can always make use of characteristics of an individual data server specification, that bears most advantages for a given configuration.

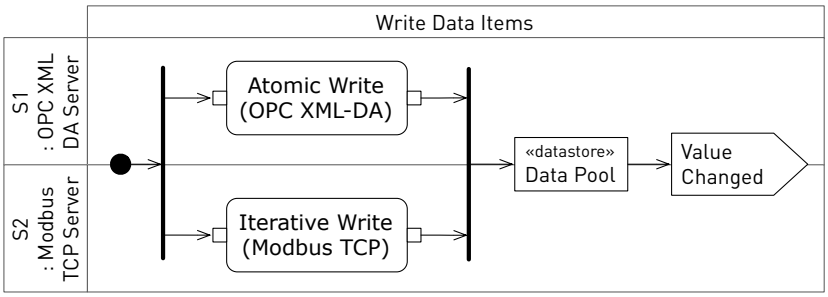
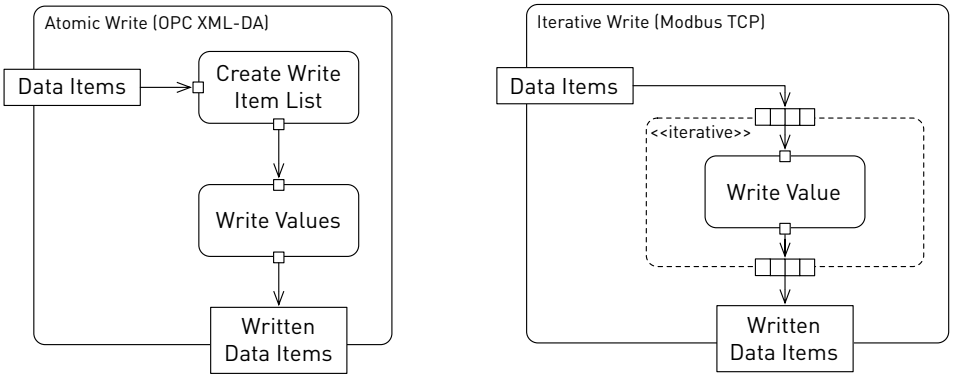


Figure 3.28: Based on the model depicted in Figure 3.27, a transformation is responsible for making use of the most suitable characteristics: A Write Data Item Effect tries to realize an atomic write operation if it is provided by the particular data server specification. Otherwise, it uses a fall-back strategy: While the OPC XML-DA specification proposes to write all data items with a single request (comp. Figure 3.29a), a Modbus TCP data provider writes the data items iteratively (comp. Figure 3.29b).

In this case, the runtime solution divides the write operation into two parallel actions, as depicted in Figure 3.28. The OPC XML-DA specification [OPC Foundation, 2004] provides a *Write Item List* method, which takes a list of data items to be written to the process in a single request. Thus, it realizes an atomic write operation (see Figure 3.29a). The Modbus TCP specification, however, does not provide such a method per se. Hence, it has to be recreated using an iterative method, as depicted in Figure 3.29b. Forcing to write several data items in an iterative manner, independently of the individual characteristics, can be realized using the means of the Algorithm Model (comp. Section 3.2.1). Particularly, Write Data Item Actions can be connected in the required order using Control Flows.



(a) The OPC XML-DA specifation provides a method for atomic write operations.

(b) Fall-back strategy for Modbus TCP servers: Data items will be written iteratively.

Figure 3.29: Strategies for writing data items; each strategy is in turn the consequence of the limitations of an individual data server specification.

3.5 Conclusions

To meet Requirement 1 on page 22, this chapter has thoroughly analyzed the relevant target domain and formalized it into a domain specific *Language Model* that is capable of capturing the operative characteristics of visualization solutions for supervising technical processes. Meeting the requirements of existing human machine interfaces in automation by working out modeling elements as already provided by existing visualization systems have mainly shaped this language model. It also considers the recommendations of the guidelines [VDI/VDE, 1997 – 2005]. However, it is not expedient to provide building blocks for each foreseeable requirement, as this would raise the complexity of the modeling language to an extent where it is hardly maintainable. Hence, two important extension points were introduced that neither require to enhance the metamodel nor modify the tool environment to include future requirements: (1) the Boundary concept of the Algorithm Model and (2) the Generic Server element of the Client Data Model. The Boundary concept, however, should be used with care: Even though it is an important and powerful extension point, enabling to integrate virtual any functionality, it requires to pollute the pure functional model with platform specific aspects. The Generic Server remains the functional model clear for platform aspects, but it prevents modelers to capture variable operative characteristics using their respective terminology.

Additional language constraints, defined on the core language model, were

necessary to capture the target domain as precise as possible. Defining these invariants is not limited to meet the requirements of the target domain. It can also be used to reduce possible modeling errors through model integrity checks of any complexity and, thus, to prevent these errors to be propagated to the final source code. It is the sufficient condition for meeting Requirement 3 on page 25. Furthermore, the more precise and comprehensive those model integrity checks are, the less complexity has to be wielded when maintaining transformations.

A precise language behavior definition must not only be provided because it is important for modelers to know certain aspects about the runtime behavior when capturing variable operative characteristics, but also as precondition for meeting Requirement 4 on page 26. Sun et al. (2008) state that if the behavioral semantics are expressed in a formal way, the transformations can also be verified whether they meet the defined semantics. Kleppe (2009) recommends to use operational or translational approaches to express semantics. Although operational semantics are more precise and thus allow for verifying transformations, they are, according to Völter (2009), not "sufficiently pragmatic to be useful in mainstream DSL practice". Consequently, it is argued that the translational approach is more sufficient to increase the acceptance of abstract user interface modeling approaches in industry. This chapter uses object and activity diagrams as well as precise textual descriptions to explain translational semantics by means of a small but representative scenario.

With this language model — the core language model, the language model constraints, and the language behavior definition —, a precise modeling of the operative characteristics of user interfaces for monitoring and operating technical processes is possible. Hence, it forms the basis for automatic transformations to the source code of different target platforms, as required by Requirement 4 on page 26. Another possibility is to interpret the model by a (meta-)program or a certain runtime environment, as mentioned by Völter (2009). The interpreter "reads the model and executes code (calculations, communication, UI rendering) as it queries or traverses the model" [Völter, 2009]. In either case, modelers can concentrate on the operative characteristics by specifying *what* a particular runtime solution is supposed to be about. The transformation (or the interpreter) defines the *how*. It takes care of specific underlying techniques and is responsible for always choosing the most suitable realization alternative.

Chapter 4

Concrete Syntax Notation

A *Concrete Syntax Notation* is built atop of the *Language Model* and, thus, constitutes the interface between a modeler and a model. More precisely, it defines a specific notation for the modeling elements and with it a tool to express and to maintain models. Stahl et al. (2007) state that it is a crucial design goal of a concrete syntax notation to enable modelers to work and to think within their particular domains. Technical conditions or limitations should not influence the design process [Stahl et al., 2007], as the concrete syntax notation is vital for an efficient employment of the modeling language and, thus, for its acceptance.

Dividing the domain *Visualization Application Model* into the sub domains (1) Algorithm Domain, (2) Client Data Domain, and (3) Presentation Domain, as discussed in Chapter 3 and depicted in Figure 3.7a, was, among others, necessary to be able to provide an individual and most expedient concrete syntax for each (sub-)domain. Consequently, modelers can think and work most comfortably in each associated (sub-)domain. This chapter works out a concrete syntax notation for the *Language Model* deduced in Chapter 3. By this means, each subdomain is treated individually.

4.1 Preliminary Consideration

Section 2.4 distinguishes between graphical and textual concrete syntax notations. A design goal of central importance is, as stressed in Section 2.4, to keep the concrete syntax notation — whether textual or graphical — as simple and intuitive as possible. A good cognitive fit to the respective thinking style leads to an effective and efficient problem-solving process [Ottensooser et al., 2012]. Kelly and Pohjonen (2009) point out that "[p]eople recognize things by their shapes, not by labels" and, thus, the symbols of visual modeling languages must neither be too simple nor too complex. They propose "that the best symbols are pictograms rather than simple geometric shapes or photorealistic bitmaps". The same applies to text-based notations, as they should be concise but expressive [Völter, 2009] and

understandable.

Regardless of having either a textual or a graphical notation, the intention of a concrete syntax notation is that language users — domain experts, not software engineers — do not face the complexity of the *Language Model* for the sake of convenience. Hence, Fowler (2011) recommends to use common and known concepts that were established within the vocabulary of the language user: for instance, "//" for comments, "{" and "}" to build hierarchies in textual notations.

Consequently, the challenge is to deduce a language notation, whose symbols reflect common concepts of the domain, being as expressive as possible while remaining as concise as possible. Furthermore, in case of graphical concrete syntax notation, the symbols must be conveniently usable also on whiteboards, as stressed by Starr (2003).

4.2 Algorithm Model: Pure Graphical Syntax Notation

As an Executable UML realization, the Algorithm Model is expected to use the graphical terminology imposed by UML to comply with known concepts and conventions, as proposed in Section 4.1. Namely, UML defines a well-established graphical concrete syntax notation among others for its class diagrams and its state machines. However, it does not define a clear and unambiguous concrete syntax notation for all of its actions. Most of the actions share the same symbol: a rounded rectangle. Consequently, common Executable UML tools established text-based, so-called *Action Languages*, such as the *Object Action Language* (OAL) [Mentor Graphics, 2008] in *BridgePoint* [Mentor Graphics, 2011] or the *Action Specification Language* (ASL) [Wilkie et al., 2003] in iUML [Abstract Solutions, 2011]. Due to this proliferation, the OMG has introduced the *Action Language for Foundational UML* (Alf) [OMF, 2010]. Not only is it a text-based concrete syntax notation for an *Action Language*, but it is also an independent language specification that is built atop the FUML semantics domain.

All of the aforementioned textual approaches are independent languages (DSLs as well), defining their own abstract and concrete syntax. Even though their concepts can be mapped to the concepts of FUML (Alf explicitly proposes this), this entails a model-to-model transformation (see e.g. [OMF, 2010, p. 329]). Consequently, two metamodels are required to be maintained. Furthermore, the procedures expressed with one of these textual action language must be integrated into the Executable UML model through M2M-transformations. This also requires to maintain additional transformations.

Irrespective of this, text based notations are inherently one-dimensional, as

pointed out in Section 2.4. UML actions, however, are connected to each other with data or control flows and can be executed on parallel branches. A modeler might capture these relationships more reliably using graphical concrete syntax notations. Starr (2003) additionally states that "[f]lows replace the need for creating, naming, and scoping temporary variables". He proposes the graphical action language *Starr's Concise Relational Action Language* (Scrall). Due to the mentioned advantages, a pure graphical notation is added to the Algorithm Model. Even though Scrall is compatible to UML, it is rather a pragmatic interpretation of the UML action metamodel with its own execution semantics specification. Nevertheless, its terminology forms a sound basis to create a graphical concrete syntax notation.

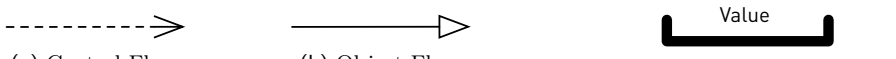
To provide a graphical terminology for the Algorithm Model that is based on the one of UML, three variants can be distinguished: (1) Using symbols that already exist unambiguously in the UML specification such as the *Send Signal Action* symbol, (2) improving existing symbols due to their inexact expressiveness, or (3) introducing new symbols.

Using Existing Symbols

UML already defines well-known symbols for the modeling elements of its class and state machine diagram. The following actions already have a clear terminology defined: (1) Send Signal Action [OMG, 2010, p. 291], (2) Accept Signal Action [OMG, 2010, p. 242], (3) Control Node [OMG, 2010, p. 368], and (4) Expansion Region [OMG, 2010, p. 380]. Hence, there is no need to change these symbols.

Improving Existing Symbols

The graphical notations of Control Flows and Object Flows are already defined in UML. However, both modeling elements share the same symbol — a solid arrow. Even though both elements can be identified in a diagram according to their connection points, a clear distinction cannot be made at a first glance. Hence, the symbols shown in Figure 4.1a and Figure 4.1b were introduced.



(a) Control Flow (b) Object Flow (c) Value Specification Action

Figure 4.1: Improved UML actions, mainly adopted from the symbols of the *Scrall* language specification [Starr, 2003].

Furthermore, Figure 4.1c presents the improved symbol of the graphical notation of a VALUE SPECIFICATION ACTION. It can be seen as a *tray* from which the value can be taken when executing the procedure. So far, UML only defines a rounded rectangle for these actions, whereas its label represents the actual value. The same symbol is intended for arbitrary actions. Thus, it might lead to confusion.

Introducing New Symbols

The third variant to provide a graphical concrete syntax notation for the ALGORITHM MODEL comprises of creating new symbols: On the one hand, new modeling elements were introduced in Chapter 3 and on the other hand, the notation of particular elements is out of the scope of UML.

Newly introduced elements of the class perspective are the BOUNDARY and the STRUCTURED TYPES. As the meaning of the latter one can be seen as specialization of a CLASS, it is depicted by the class symbol with the keyword *«Structured Type»*. The BOUNDARY is meant as a *black box* hiding functionality that is beyond the actual system. Hence, its symbol is presented as a rectangle — the box — with the BOUNDARY'S name as its label. It contains both the REQUIRED INTERFACE and the PROVIDED INTERFACE (both depicted using UML's interface notation, a classifier symbol with the corresponding keyword: *«Required Interface»* or *«Provided Interface»*).

Creating new symbols for UML actions has mainly been inspired by the symbols of the *Scrall* language specification [Starr, 2003]. Hence, Figure 4.2 shows UML actions with symbols that were reused from Scrall without modifications.

(a) Create Link Action (b) Destroy Link Action (c) Destroy Object Action

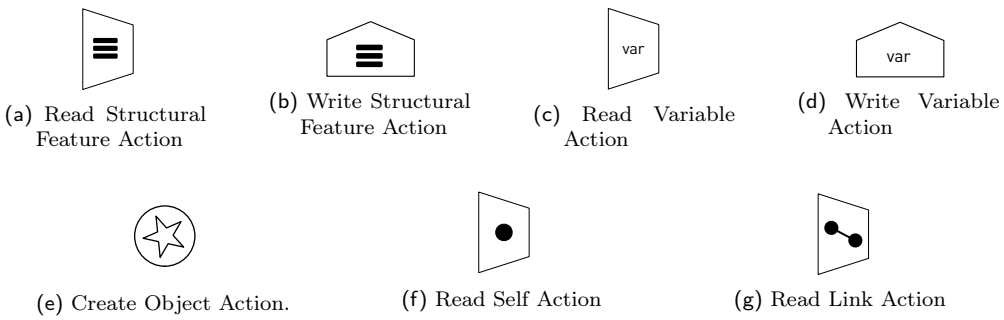
Figure 4.2: Actions with their symbols that were taken over from the *Scrall* language specification [Starr, 2003] without modifications.

Before creating new symbols for UML actions, two intrinsically different kinds of actions must be distinguished: the *Read** and the *Write** actions[1]. For the sake of keeping the language notation as concise as possible, each of both action types will make use of the same surrounding shape. As read actions are used to select *something*, their surrounding shapes look like funnels (e.g. see Figure 4.3a). Write

[1]The character "∗" acts as a wildcard.

actions are used to put a value *to somewhere*. Thus, they are surrounded by a shape similar to an *elevator cabin* (e.g. see Figure 4.3b).

Figure 4.3 shows the new symbols for the UML actions that lack a concrete syntax notation so far. Structural Feature Actions, as depicted in Figure 4.3a and Figure 4.3b, gain access to class attributes. Hence, their symbols indicate a property list of a class. Variable Actions, shown in Figure 4.3c and Figure 4.3d, are distinguished using the keyword *var*, as it is used for variable declaration in many programming languages. Figure 4.3e shows the symbol of a *Create Object Action*. Similar to its counterpart Destroy Object Action, as introduced in Scrall and presented in Figure 4.2c, it is surrounded by a circle. The embracing star indicates that it gives "birth" to an object. Figure 4.3f shows the symbol of the Read Self Action. It is a *dot* indicating the actual object, whereas the symbol of the Read Link Action, shown in Figure 4.3g, indicates a link between two objects.



(a) Read Structural Feature Action

(b) Write Structural Feature Action

(c) Read Variable Action

(d) Write Variable Action

(e) Create Object Action.

(f) Read Self Action

(g) Read Link Action

Figure 4.3: UML actions with their respective newly created symbols.

Finally, Figure 4.4 presents the symbols of the actions that were introduced in the Algorithm Model as extension of the UML specification. The symbols of the Data Item Actions, shown in Figure 4.4a and Figure 4.4b, indicate a point of measurement in the process, as specified in [DIN, 1993]. As shown in Figure 4.4c, a Read Boundary Action (comp. Section 3.2.1) contains a symbol that implies two interfaces to a system whose concrete realization is not relevant at time of modeling. Figure 4.4d presents the symbol of a Read Presentation Model Action. It indicates a window of a user interface containing a trend chart.

(a) Read Data Item Action

(b) Write Data Item Action

(c) Read Boundary Action

(d) Read Presentation Model Action

Figure 4.4: Symbols for the actions introduced as extension to the UML specification, as discussed in Section 3.2.1.

4.3 Client Data Model: Pure Textual Syntax Notation

Depending on the actual technical process, hundreds or thousands of process variables (Data Items, see Section 3.2.2) might have to be configured through the Client Data Model. Composing the data model and the required communication relationships, consequently, entails the need to express a huge set of structured configuration data. Neither comprehensive data hierarchies nor relationships that are only conceivable using graphical connection elements are to be modeled. As lightweight textual representations have advantages over graphical ones mainly concerning scalability, their usage leads to more efficiency when modeling process data.

Conventional programming languages provide constructs to express structured data types, such as the *struct* in the C programming language (see Listing 4.1 for an example) or a *class* in Java (see Listing 4.2). As these techniques are well-known, they form the basis to create a textual notation.

Listing 4.1: C struct.

```
struct my_struct {
    // comment
    int a;
    bool b;
    char *c;
    other_struct *s;
}
```

Listing 4.2: Java class.

```
class myClass {
    public int myAttribute;

    public void setMyAttribute(int newAtt) {
        this.myAttribute = newAtt;
    }
}
```

Listing 4.3 depicts the underlying construction rule of the text based notation: Based on a root element, all model elements embrace their containing elements using "curly brackets". The definition of attributes has two facets: (1) If labeling of attributes is not necessary, then they will be written next to the name of the model element. This is mainly relevant for the Name attribute. (2) Attribute labels describe their values, whereas both are separated using a colon. This hierarchy cannot become too complex, because the *Core Language Model* only

defines three sub levels within the Client Data Model. (In case of treating each of the Logical Data Perspective, the Technical Data Perspective, and the Alarm Perspective as root elements and, thus, as individual assets, as depicted in Figure 5.7, only two hierarchy levels are available.) Furthermore, many unnecessary characters were omitted for the sake of conciseness. For instance, there is no need to close a line with a particular delimiter such as a semicolon, as required in many programming languages.

Listing 4.3: General construction rule of the text based concrete syntax notation.

```
ROOT_MODEL_ELEMENT {
  CONTAINING_MODEL_ELEMENT ATTRIBUTE_VALUE, ... {
    ATTRIBUTE_LABEL : ATTRIBUTE_VALUE
    REFERENCE_LABEL : REFERENCE_VALUE
    ...
    CONTAINING_MODEL_ELEMENT ... {
      ...
    }
  }
}
```

Listing 4.4 exemplarily shows how a Data Item can be expressed using the concepts, deduced in Listing 4.3. The root element Logical Data Perspective embraces the element Data Item (line 2), which is characterized by its Name "Fuellevel_1" and its Pointing To relationship "xmlda_Fuellevel_1". (A little verbosity was added through the term "pointing to" in between, for improving readability.) Line 4 refers to a Signal, as discussed in "Logical Data Perspective" of Section 3.2.2. An attribute of the Data Item is depicted in line 5. Line 6 shows a containing element, the Alarm Behavior.

Listing 4.4: Example configuration of the Logical Data Perspective using the concrete syntax notation.

```
1  LogicalDataPerspective {
2    DataItem Fuellevel_1 pointing to xmlda_Fuellevel_1 {
3      SignalOnChange : Fuellevel_1_Changed
4      MaxValue : 280.0
5
6      AlarmBehavior Fuellevel_1_AB_1 {
7        LimitValue : 250
8        Threshold : 5
9      }
10   }
11 }
```

Following these aforementioned concepts, the concrete syntax of the entire Client Data Model was settled using a lightweight text notation.

4.4 Presentation Model: Combined Syntax Notation

For creating graphical user interfaces, whether they are industrial or conventional, almost all existing development environments allow to graphically arrange user interface elements on their workspace. Thus, they provide a visual feedback to the developer. In its simplest form, only placeholder images are added to and dragged around on these tool's engineering workspace. More sophisticated environments, on the contrary, allow to more authentic user interface engineering, as they populate the components on the workspace with static, yet real data.

These facts apply to common interface builders as well as to engineering tools dedicated to create visualization solutions in industrial automation. As a consequence, an efficient concrete syntax notation of the Presentation Model is graphical and assigns at least appropriate placeholder images to the individual UI Component elements (comp. Figure B.21). It is up to a proper tool to transfer the position coordinates and the size parameters — both being gainable from the tools workspace — into the model to populate the Position and Size properties of the particular UI Component's Representation. Indeed, it can be a burden for modelers to configure each parameter of a UI Component using only such high-fidelity concrete syntax notations. Besides the Position and Size parameters, UI Components contain further Representation properties and, in addition, Animation and Interaction properties as well. Consequently, a concrete syntax that allows the combination of both graphical and textual assets is most expedient in case of the Presentation Model. In addition, this choice bases on the fact that the Presentation Model contains the Color Definition, Image Bundle, and the Multi Lingual Text Definition elements. Particularly the latter one is comprised of many text items having no relationship to each other. Figure 4.5 depicts a simplified excerpt of the metamodel of the Presentation Model, as defined in Figure B.21 and Figure B.22. It highlights which kind of concrete syntax notation is added to which part of the Presentation Model, so as to provide an efficient and convenient tool for modelers — the domain experts.

The graphical concrete syntax notation distinguishes between (1) the *Navigation Subsystem*, and (2) the *Panel Subsystem*, as shown in Figure 4.5. The *Navigation Subsystem* behaves much like a *state machine*, whereas the Panels are treated as *states* and the Navigation Flows as *transitions*. Each *state* conceals a *Panel Subsystem* in the navigation subsystem, which in turn provides means for arranging UI Components, as illustrated in Figure 4.6. The textual concrete syntax notation, on the other side, makes use of the concepts that were deduced in Section 4.3: It provides a syntax which is very similar to the constructs of well-known programming

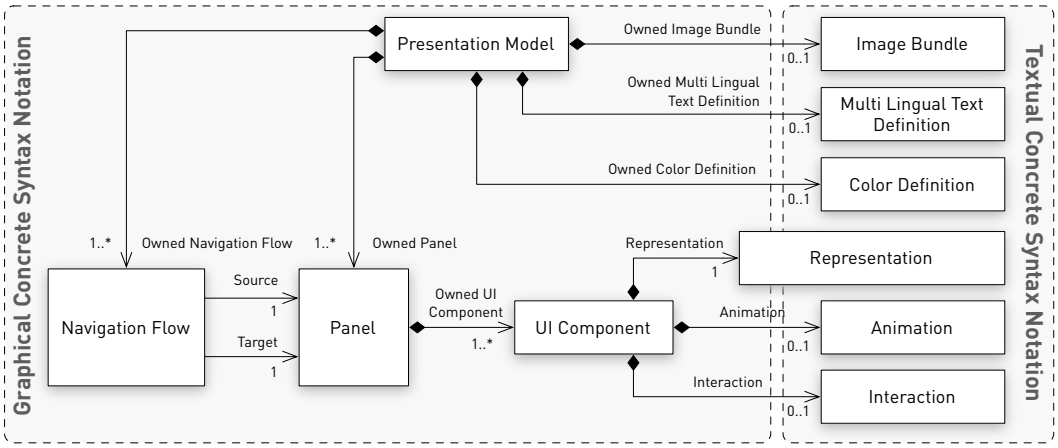


Figure 4.5: Contrasting both the graphical and the textual modeling assets of the Presentation Model.

languages to be used to express structured data types (comp. Listing 4.1 and Listing 4.2), as defined in Listing 4.3.

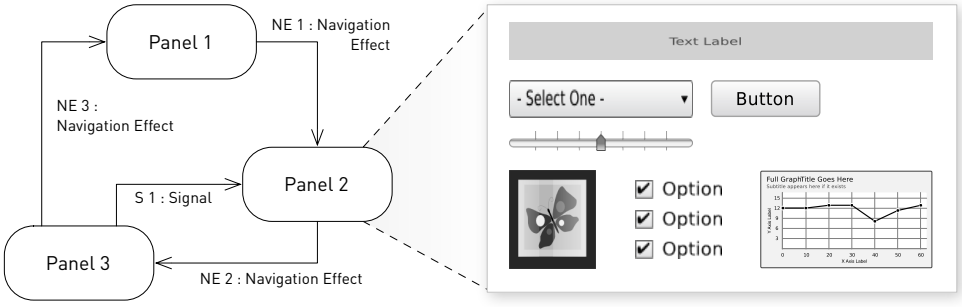


Figure 4.6: Graphical concrete syntax notation of the Presentation Model: The *Navigation Subsystem* (left hand side) looks very much like a *state machine*. Each state conceals a *Panel Subsystem* (right hand side), which provides high-fidelity user interface modeling.

4.5 Conclusions

Authors of DSL literature — Fowler (2011); Stahl et al. (2007); Strembeck and Zdun (2009); Völter (2009) — agree that a concrete syntax notation is of central importance. It is the interface between the modeler — the automation engineer

as domain expert — and the model itself. Hence, successful modeling requires an expressive abstract syntax as well as a concrete syntax that has been tailored to the qualifications and experiences, but also to the habits of the domain experts. Völter (2009) points out that it is a misconception to "convince domain users or experts about a 'better notation' — just implement what they have". That is precisely what was the intention of this chapter: To work out a concrete syntax notation for the *Language Model*, discussed in Chapter 3, that is close to the problem to be solved, as stressed by Requirement 2 on page 24. This is a precondition for reducing the complexity of the modeling language and raising efficiency during modeling and with this for getting accepted by modelers.

Particularly, the proposed concrete syntax makes use of graphical and textual notations, in a way that meets the aforementioned requirements best. The graphical notation of the Algorithm Model uses the common UML terminology and introduces new symbols with respect to the UML actions. Starr (2003) requires graphical symbols to easily be drawable on whiteboards. Since the introduced symbols use simple geometrical forms, this requirement is met. Even though the icons of the individual actions (see e.g. Figure 4.4) are more or less complex, they can be replaced by simpler characters on whiteboards. For instance, the letter "B" can be used to indicate the Read Boundary Action instead of the respective symbol when sketching[2]. The graphical part of the Presentation Model uses a state machine like notation for modeling the navigation hierarchy through mimics. An individual Panel allows to compose *UI Components* using respective placeholders, and herewith foster the comprehension of user interface designs. The state machines as well as the visual feedback tool are common concepts in the automation domain and, thus, comply with modelers' expectations.

Textual notations were introduced for the Client Data Model and determined to be parts of the Presentation Model. Even though it is required to express structured parameters, they have little or no dependencies among each other. Hence, this concrete notation is very similar to the *struct* construct of the C programming language — a programming language that automation engineers learn in the first semester of their studies. However, remaining clarity and expressiveness for textual representations, even if the model has grown to a large extend, is an important requirement for an appropriate DSL tool. It can support the modeler with syntax highlighting, folding, and auto-completion.

To conclude, Stahl et al. (2007) state that the availability of tools and concepts must not have an impact on the design of the concrete syntax notation to keep the

[2]This is not appropriate for modeling, as letters can be ambiguous: "L" can mean Read Link Action or Write Link Action.

design process free of any reasoning about technical conditions. The modeler is the central focus. However, this only works if the particular modeling language is not a part of an existing modeling environment, as it is the case in this thesis. After having defined the concrete syntax, the next chapter will discuss the technical framework.

Chapter 5

Movisa: Domain Specific Modeling Workbench

Providing an efficient and easy-to-use tool is a key factor for the acceptance of new development methods, whether they are model-driven or conventional. This chapter is about evaluating existing approaches and tools that can be combined to assemble the concepts of Chapter 3 and Chapter 4 into a tailor-made modeling workbench. It is anticipated that the *Eclipse* framework [Eclipse, 2011b] represents an extensible, open, integrative, and, thus, powerful foundation to realize these concepts. Section 2.4 introduces Eclipse plugins to create both and graphical concrete syntax notations. Section 2.3 points out that different model transformation engines are available within this framework. But more importantly, Eclipse is an accepted tool framework also in automotive and aerospace engineering (e.g. see [TOPCASED, 2011]).

Consequently, the *Core Language Model* was realized with *EMF* and *Ecore*, as a foundation for the other language aspects: namely, the *Model Verification*, *Model Transformations*, and, finally, the *Concrete Syntax Notation*. Each of these components comes as an individual Eclipse plugin. Together, they form a tool enabling modelers to simply create, use, and maintain models of the language, deduced in Chapter 3, as well as to transform them into a particular target technology. This tool was made available with the following name:

> **Movisa**
> *Model Driven Development of Visualization Solutions in Industrial Automation*

The following presents best practices from the industry that were taken into account as requirements when realizing the language concepts in a tool. After that, the realization of specific characteristics of the language model will be discussed.

5.1 Requirements

Völter (2009) proposes a set of best practices for language design that serve as a good starting point to define requirements for implementing the *Language Model*, deduced in Chapter 3, and the *Concrete Syntax Definition*, that were contributed in Section 5.5. These requirements are the following:

Requirement 5.1 (Checks first and separate)**:** *This requirement addresses the verification of the wellformedness of models, based on the* Language Model Constraints *(comp. Section 3.3). Völter (2009) sees it necessary to treat these constraints "as first class [DSL artifacts having] their own phase during model processing". Although language model constraints can be checked during transformation, it would increase the complexity of the transformations. Hence, the wellformedness of models should be proven as early as possible in the design process. Attaching descriptive error messages to the constraints makes them even more powerful. This requirement refines Requirement 3 on page 25.*

Requirement 5.2 (Care about templates)**:** *Since the aim of the proposed approach is to introduce sustainability due to short innovation cycles of the used platforms, transformations are required to be constantly under development so as to keep existing solutions working on new platforms. An aspect of reducing development efforts concerns the maintenance of the modeling language when it is in use. With this respect, Völter (2009) recommends to use the same development techniques for transformations as they are used in general software engineering: They should be treated as one of the central artifacts of the domain specific tool environment, because they include all the knowledge about how to map the domain concepts to the runtime artifacts and particularly, they define the* Language Behavior*. Consequently, reasonable modularization techniques are recommended.*

Requirement 5.3 (Control manually written code)**:** *The* BOUNDARY *concept, as discussed in Section 3.2.1, makes this best practice relevant. It allows to include arbitrary functionality using platform specific source code. For this reason, although this manually written source code resides at a final and platform specific abstraction level, it is part of the model that has to be treated as a operative characteristic.*

Requirement 5.4 (Tooling Matters and Model Partitioning)**:** *A crucial aspect to accept new methods in industry is the availability of powerful tools that hide at least most of the complexity of these new procedures and, thus, lower the initial hurdles for developers. These tool environments, however, must be compatible with established expectations and workflows such as versioning, syntax highlighting, and*

support for teamwork; visualization solutions are projects involving experts from different domains. Ensuring that several modelers are able to simultaneously work on models can be achieved with a good partitioning strategy, as mentioned by Völter (2009). Reasoning about how to physically split a model into smaller, but connected parts, might also have an impact on scalability, as a tool does not have to load the entire model completely [Kolovos, Paige, and Polack, 2009].

5.2 Model Verification

Section 3.3 exemplarily presents numerous constraints that must be met by modelers to keep their models wellformed. Particularly the *model-integrity checks* and their associated constraints can be of any complexity; for instance, also between elements residing in different submodels. Against this background and to tackle Requirement 5.1 on page 92, Raneburger et al. (2011b) propose a technology stack with three successive stages to ensure wellformedness of models: (1) model constraints imposed by the metamodel, (2) OCL constraints, and (3) Java constraints. Figure 5.1 shows that models are wellformed and of integrity if all kinds of constraints were verified to be fulfilled.

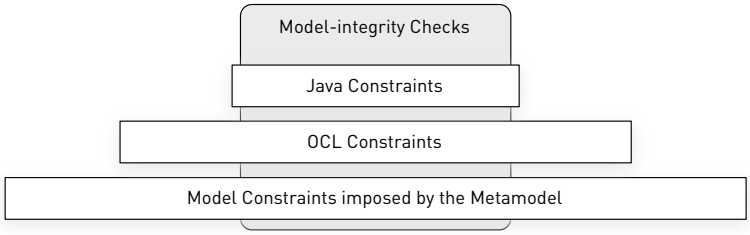


Figure 5.1: Technology stack to provide model-integrity checks, as proposed by [Raneburger et al., 2011b, Figure 1].

The consequence of this consideration is the requirement of a flexible verification engine, particularly for combining OCL constraints and those constraints expressed in a general-purpose programming language. Both the *Epsilon Validation Language* (EVL) [Eclipse, 2011g] and *oAW's Check Language* [oAW, 2011] are very similar to OCL, but unlike OCL they allow side-effects. Additionally, they are able to transparently integrate constraints that were expressed in Java. Hence, they are promising candidates to express arbitrary constraints. The choice for realizing the model-integrity checks, however, fell on EVL, as it additionally provides so-called *Quick-fixes*. With their help, many constraint violations can be automatically fixed

and, therewith, an additional level of efficiency and convenience for the modelers can be provided.

5.3 Model Transformation

Section 3.4 explains the behavior of the modeling language by means of demonstrating sample configurations using UML object diagrams and showing the resulting runtime behavior using UML activity diagrams. Based on such a behavior definition, Fowler (2011) proposes to interpret the model on the target machine and only if this is not possible due to limited resources or the need for high-performant source code, transformations should be used beforehand. Interpreting the visualization solution model, however, requires appropriate runtime environments which were identified as drawback of existing approaches. Consequently, this section discusses the deduction of pertinent transformations to establish the transition between modeling elements and runtime artifacts realizing the language behavior while tackling towards meeting Requirements 5.2 and 5.3.

5.3.1 Dimensions of Code Generation

Providing model transformations means to make a meta-program — a program that assembles other programs — available. On the one hand, code generation tools need the same development methods as other programs. On the other hand, Kleppe (2009) states the significant challenge these tools are faced with: The life span of a code generator is usually short, as the modeling language underlies a certain evolution (as Völter (2009) explicitly claims). Even the target language will change due to new platforms and devices or innovations on existing ones. Consequently, the transformation process always needs to be adopted to new requirements. The following summarizes some facts about code generation that were assembled from [Czarnecki and Helsen, 2003; Fowler, 2011; Kleppe, 2009; Mens, Czarnecki, and Gorp, 2006; Stahl et al., 2007; Völter, 2009] to make adequate decisions when deducing a transformation tool.

Transformer Generation: A piece of software that navigates the input model and, based on this, creates the output text. This method is particularly useful if most of the output text is generated.

Templated Generation: If the output text mainly contains static text, then template based methods are more efficient, as they allow to combine both of these static parts and required statements to include the dynamic aspects from models into so-called *templates*.

Model-Aware Generation: This dimension of code generation has an impact on the kind of organizing of the runtime artifacts. In this case, models are used to create configuration code in order to instantiate a reference implementation. This method simplifies the development of the code generator, as the logic to handle the models is encapsulated in the reference implementation. However, it possibly demands more resources at runtime.

Model-Ignorant Generation: Particularly if the target environment has limited resources, it might be necessary to create target code that does not explicitly replicate the particular model. Hence, this method demands to encapsulating all logic into the code generator.

Generation Gap: A challenge of model-driven software development is to treat both generated and manually written code. It must be ensured that handwritten code will not be affected if the model is regenerated.

Horizontal vs. Vertical: This dimension distinguishes if source and target model reside at the same level of abstraction (horizontal transformation) or at different levels of abstraction (vertical transformation). See also Definition 2.3 on page 13.

Ability to guarantee correctness: If the transformation creates wellformed target artifacts, Mens, Czarnecki, and Gorp (2006) call this *syntactic correctness*.

Based on these dimensions, some characteristic aspects of the language model will be discussed in the following.

5.3.2 Specific Characteristics of Code Generation

The language model of Chapter 3 makes use of different concepts which need different code generation strategies to be applied. Based on the aforementioned dimensions of code generation, representative characteristics of the language model are discussed against the background of generating runtime artifacts.

UI Components

In every target environment, the code to provide a graphical user interface contains many artifacts and statements which are independent of the input models. As fixed operative characteristics, they only serve to establish the required infrastructure, e.g. with respect to applied design pattern. Furthermore, a variety of assumptions were made about the standard behavior and appearance of UI COMPONENTS while deducing the language model in Chapter 3. For instance, besides the configurable parameters, each ALARM CONTROL widget provides means to sort and filter alarms, as discussed in "Alarm Perspective" of Section 3.2.2. Consequently, a *templated*

generation in combination with *model-aware generation* is most expedient to translate this particular kind of modeling data — the UI COMPONENTS — into runtime artifacts.

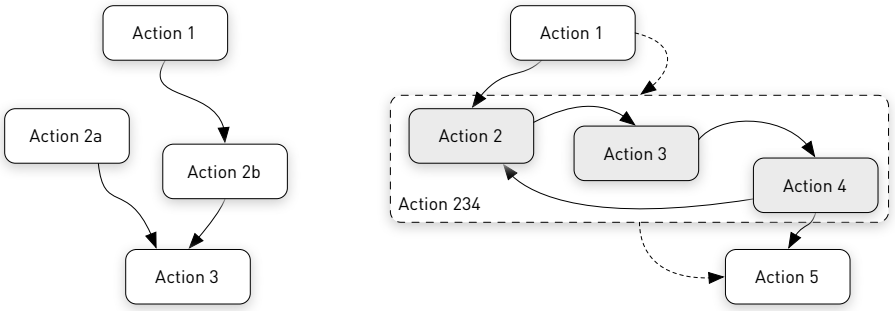
Boundary

The BOUNDARY was introduced to integrate custom functionality into a VISUALIZATION APPLICATION MODEL using platform specific source code. Stahl et al. (2007); Völter (2009) recommend to strictly separate handwritten and generated code. However, this might not always be possible due to restrictions in the target language or even in the target platform. As a compromise, template languages such as the *Epsilon Generator Language* (EGL) [Rose et al., 2008] or *Xpand* [oAW, 2011] are able to mark specific regions in the generated source code as *protected*. These regions will be preserved by every subsequent transformation, as required in Section 5.3.1 (close the generation gap) and to meet Requirement 5.3 on page 92.

State Activities

ACTIONS within a STATE ACTIVITY BLOCK are connected through control or object flows. Hence, these modeling elements establish *directed graphs* which can be analyzed using graph algorithms: The correct order to treat these actions has to be ensured so that the preconditions of each action can be fulfilled. For instance, when creating the particular runtime artifacts of the actions depicted in Figure 5.2a, the runtime equivalents of "Action 2a" and "Action 2b" have to be considered before the ones of "Action 3". This can be ensured using the *topological sort* algorithm, as presented by Cormen et al. (2001). However, the *topological sort* algorithm cannot handle *cyclic directed graphs*, as shown in Figure 5.2b. These cycles in the action model indicate loops that are performed at runtime. Hence, specific methods for the runtime artifact creation are required. According to Tarjan (1972), vertices in a directed graph, which form a cycle, are called *Strongly Connected Components*. They can be identified using *Tarjan's Algorithm*. This consideration leads to the following steps which are performed by a transformation:

(1) Identify strongly connected subgraphs.
(2) Treat each strongly connected subgraph as a single component, as shown in Figure 5.2b.
(3) Perform a topological sort.
(4) Create for each component the particular runtime artifacts and resolve the strongly connected components through appropriate flow control means. For instance, the "Action 234" will be resolved to a *while loop* with "Action 2" as



(a) Topological Sort: "Action 3" requires both "Action 2a" and "Action 2b" to be treated beforehand.

(b) Strongly Connected Component: The solid arrows indicate the actual control flow, whereas "Action 2, 3, 4" establish a cycle. To apply the *topological sort* algorithm, these three actions need to be treated as a single one, as the dashed action and arrows indicate.

Figure 5.2: Demonstrating the fundamentals of the used graph algorithms: (a) shows an *acyclic directed graph*; (b) presents a *cyclic directed graph* (solid arrows) as well as the required substitution to become an *acyclic directed graph* (dashed action and arrows).

loop entry and "Action 4" for evaluating the exit condition.

Concluding, executing these very specific algorithms needs powerful transformation methods. For instance, they could be realized with general purpose programming languages such as Java or C++ (*Transformer Generator*). Furthermore, *model-ignorant* generation possibly leads to more performant runtime artifacts.

Client Data Model

Runtime artifacts that ensure reliable communication relationships with different data servers mainly contain fixed operational characteristics — source code that is independent from the input models. (This is very similar to the generation of the runtime artifacts providing the graphical user interface.) For instance, source code that assembles SOAP [Gudgin et al., 2007] messages, required for message exchange, will be the same for every model. The actual process variables to be transmitted using these SOAP messages in turn characterize an individual model. This information can be translated to runtime artifacts using *model-aware generation* techniques using *templates*.

CUI-to-CUI Transformation

For deploying a user interface model to a *hardware platform*, that differs from the one for which the initial user interface model has been created, the CAMELEON REFERENCE FRAMEWORK (comp. Section 2.2) proposes to consider the user interface model in a new *context of use*. Thus, it proposes to *translate* it to this new context of use: a *CUI-to-CUI* transformation is required (comp. Figure 2.2). Particularly, this transformation modifies the user interface model so that it complies with the given hardware characteristics, such as screen size or interaction means. In case of the screen size attribute, a first step for adaption is to scale all user interface elements using a certain factor. However, this approach has limitations, as the user interface might become not readable and, thus, not usable. Hence, the second step is to reorder or even to restructure the user interface.

Besides the *platform* aspects, a context of use is, according to Calvary et al. (2003), determined by *user* and *environment* properties. Even though transformations are able to translate a user interface model into another context of use when having these information at their disposal, the level of complexity would be raised: Both a comprehensive amount of information and transformations considering all these factors are necessary. Meskens et al. (2009) point out that an automatic context-of-use adaption can lead to user interfaces that behave differently than they are supposed to. This in turn might lead to hazardous situations due to operating errors. Consequently, the strategy pursued in this thesis is to shape the transformation process semi-automatically, as is a more pragmatic but safer approach. The modeler is in charge of manually configuring the transformation process beforehand and of tailoring the transformation result afterwards: (1) By defining coherent presentation units (comp. Definition 3.1 on page 51), an immutable configuration will be ensured, as these units are not torn apart. (2) Instructions for the transformation are to be expressed in the original user interface model through model annotations so as to provide a certain degree of semantics to the UI COMPONENTS. Table 5.1 presents supported annotation properties together with their meanings.

Table 5.1: Supported model annotation properties.

ANNOTATION PROPERTY	DESCRIPTION
Top-level annotation properties.	
Scale Only	A *top-level* annotation telling the transformation only to scale all UI COMPONENTS to fit into the new context of use.
UI COMPONENT annotation properties.	

ANNOTATION PROPERTY	DESCRIPTION
On Screen Big *On Screen Medium* *On Screen Small*	Using the values *keep*, *remove*, and *text*, it can be defined whether the annotated UI COMPONENT is to be kept on the particular platform or needs to be removed. For instance, an image representing a pipe between two tanks provides certain information when operating the technical process on a *big screen* in a control room. It is, however, not necessary when checking the fill levels of the tanks from a mobile device (*small screen*). The *text* value instructs the transformation to convert the annotated component to a text component. Using the previous example, the fill levels of the tanks can be shown as colored rectangle with a *height animation*. Due to the limited screen size of mobile devices, text labels, showing the value numerically, are better suited.

Actual hardware specifications are interactively populated into the transformation through a user dialog, as shown in Figure 5.3. Based on the model annotations of Table 5.1, the *CUI-to-CUI* transformation creates a valid user interface model tailored to the given hardware characteristics. After possibly manually post-processing this model, a *CUI-to-FUI* transformation realizes the vertical transformation to create the runtime artifacts.

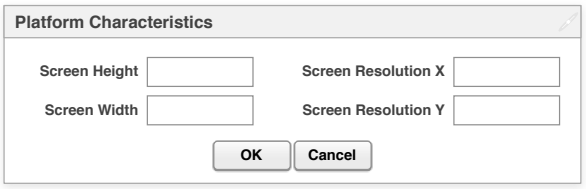


Figure 5.3: Mockup of a dialog dedicated to populate a *CUI-to-CUI* transformation with platform characteristics.

5.3.3 On the Deployment of Runtime Artifacts

Section 3.2 deduces a *Core Language Model* that is capable of capturing variable operative characteristics of visualization solutions through models. The data stored in these models form the basis for generating runtime artifacts. To comply with the language behavior, as exemplary discussed in Section 3.4, these runtime solutions must be composed in their simplest form of the following components:

Alarm Management: This component observes the configured process variables, whose values are provided by the *Data Servers*, to meet their limit values, as defined in their ALARM BEHAVIORS. If a limit value has exceeded, suitable events must be thrown to notify operators about this exception through the user interface. Additionally, "Alarm Perspective" of Section 3.2.2 introduces the *Remote Alarm Receiver* and, thus, if configured, remote events (e.g. an Email) inform remote operators.

Historical Data Management: "Logging" of Section 3.2.2 states situations where the process data must be preserved for future assessment. For this purpose, a database manages to store the possibly vast amount of data and provides an interface to it.

User Interface: This component visually reflects the operative process states through pertinent user interface widgets and enables operators to interact with the process. Furthermore, the user interface presents local alarms to operators and can be used to assess the historical data.

Furthermore, a visualization solution is required to support different data specifications which are characterized by protocols residing at different levels of the *OSI stack*. Hence, the deployment of these components strongly depends on the target platform configuration or the available technology stack. Concluding Figure 1.2, two configurations can generally be distinguished:

(1) *Sandboxed technology stack or platform with limited resources*, accompanying with the platform configurations presented in Figure 1.2a and Figure 1.2b.

(2) *Non-sandboxed technology stack or platform with ample resources*, as introduced in Figure 1.2c.

In the following, these terms as well as sample deployments of these two target platform configurations will be discussed.

Non-Sandboxed Target Platform Configuration

Realizing the aforementioned components on a platform that allows unrestricted access to underlying system functions and that has ample resources, leads to many degrees of freedom to generate the runtime solution. Figure 5.4 shows a deployment in its simplest form, where all components are executed on a single host — the visualization client.

In particular, Figure 5.4 shows a standalone realization with *Python*. As it is a general purpose programming language, it has access to the entire system API[1]. Granting the application access to the entire TCP/IP-Stack enables the

[1] Application Programming Interface

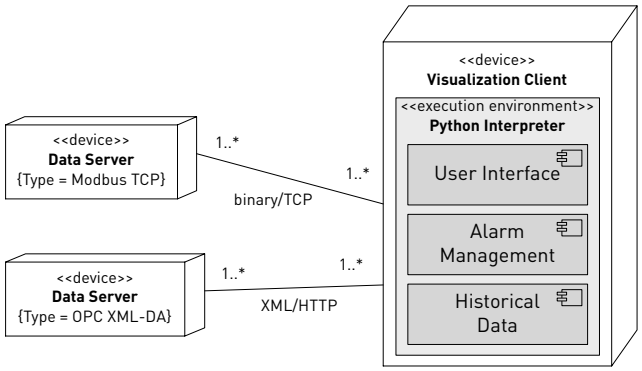


Figure 5.4: Sample non-sandboxed deployment with *Python* as target technology.

integration of the communication routines of any process data specifications right into the application, as shown in Figure 5.4. Furthermore, the platform, depicted in Figure 5.4, has enough resources to operate a database to capture historical process data (component: *Historical Data Management*). *Python* as target implementation technology provides several libraries to send Emails and to consume the *Facebook API*. Both are necessary to publish remote alarms (component: *Alarm Management*).

Sandboxed Target Platform Configuration

If the target platform does not have enough resources to operate all components or if it only provides limited access to the underlying system, which forestalls the realization of required functionality, these runtime artifacts should be transferred to dedicated processing nodes, as depicted in Figure 5.5.

Figure 5.5 presents a web-based visualization. The component *User Interface* is realized using *HTML* and *JavaScript* and, thus, it will be rendered by a web browser application. A web browser, however, is a sandbox that only allows communication relationships on the *OSI-Level 7* (e.g. HTTP[2] and FTP[3]). Consequently, not every process data specification protocol can be handled. As JavaScript applications only have limited access to system functions, they possibly cannot make data persistent. Therefore, a distributed deployment with a centralized data processing and data warehousing node becomes necessary, as presented in Figure 5.5.

[2]Hyper Text Transfer Protocol
[3]File Transfer Protocol

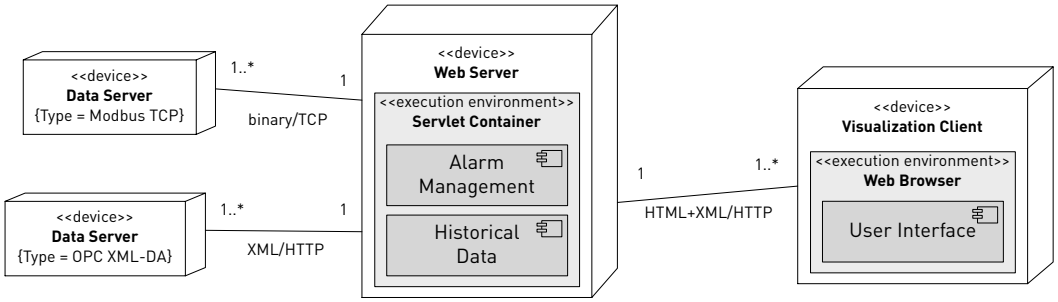


Figure 5.5: Web-based sample deployment with *HTML and JavaScript* as target technology. Because web applications are executed in a sandbox — the web browser —, certain components need to be transferred to dedicated processing nodes.

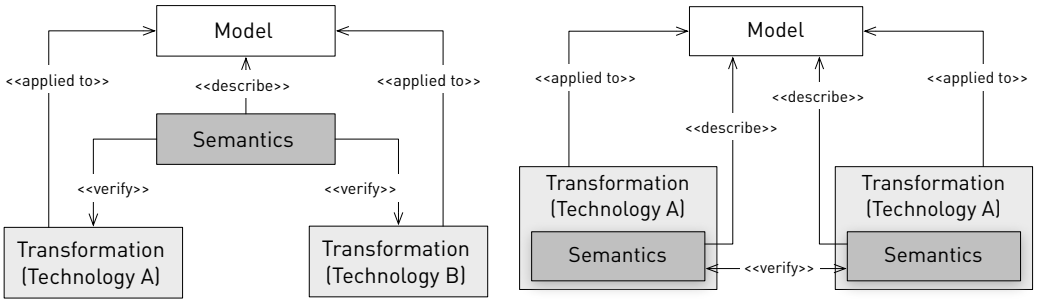
5.3.4 Evaluation of Existing Code Transformation Engines

Concluding Section 5.3.2, both horizontal (model-to-model) and vertical (model-to-text) transformation techniques are required. Only two transformation engines exist which combine these two techniques while being able to realize the particular characteristics emphasized in Section 5.3.2: the *Epsilon Tool* [Epsilon, 2011] and *openArchitectureWare* (oAW) [oAW, 2011]. *Xtend* is the M2M transformation language atop of oAW, *Xpand* provides template based M2T transformations. Although Xpand is a templated generator, it can integrate Java code for realizing a transformer, as discussed in Section 5.3.1. The Epsilon Platform provides the *Epsilon Transformation Language* (ETL) [Eclipse, 2011f] for M2M transformations and the *Epsilon Generation Language* (EGL) [Eclipse, 2011d; Rose et al., 2008] for M2T transformations. Both transformation engines are based on a common architecture: the imperative programming language *Epsilon Object Language* (EOL) [Eclipse, 2011e], that can be seen as a mixture of JavaScript and OCL. Although it enables the integration for routines written in Java, EOL reduces the need for this due to its imperative programming language nature. Hence, Epsilon is a very flexible model processing tool thus being employed in the MOVISA modeling workbench.

5.4 On the Verification of the Language Behavior

Sun et al. (2008) propose to verify transformations against an explicitly defined and formal semantics model, as discussed in Section 5.3.1 (ability to guarantee correctness) and depicted in Figure 5.6a. *Translational* approaches, as applied in

this thesis, however, contain the semantics model only implicitly. Consequently, the transformations need to be verified among themselves, as Figure 5.6b depicts.



(a) Explicitly defined behavioral semantics can be used to verify the transformations.

(b) *Translational* semantics expression implicitly contain the semantics model and thus, they can only be verified among themselves.

Figure 5.6: The concrete setup to verify the language behavior depends on how the dynamic semantics were defined.

The realization of both cases (Figure 5.6a and Figure 5.6b) in an automatic fashion, however, is not trivial: It can be verified if certain classes with required attributes will be generated. But verifying more detailed expressions requires both very precisely specified semantics models and very sophisticated verification and source code analyzation rules. Due to this effort, it is more practical to support a manual verification task with automated software tests at both the model level, as proposed by Fowler (2011), and the source code level. This thesis only considers to manually verify and test the language behavior.

5.5 Concrete Syntax Implementation

Chapter 4 sums up that it is most expedient to mix graphical and textual concrete syntax notations. The *Graphical Modeling Framework* (GMF) [Eclipse, 2011h] is the sole tool for providing a graphical concrete syntax notation to EMF based models. Therefore, the following will analyze tools to create textual concrete syntax notations against the background to integrate them into GMF.

Xtext [Eclipse, 2011i], being a framework to define textual concrete syntax notations, uses an EBNF grammar to specify the core language model and the concrete syntax. Although an Xtext model has an EMF/Ecore in-memory based representation and, thus, it can complementary coexist with GMF based editors, all editors work on the same model file. Consequently, a reasonable partitioning of the

model, as required in Section 5.1, is not possible. The same applies to models whose textual concrete syntax notation was defined using the *Textual Editing Framework* (TEF) [Scheidgen, 2008]. *EMFText* [EMFText, 2011] is a tool to map textual syntax definitions, expressed in a HUTN description (see Section 2.4), to Ecore based metamodels. Due to its modular specification means and the possibility of tailoring EMFText's reference resolvers, as discussed by Heidenreich et al. (2009), the textual syntax can simply be integrated into the GMF based graphical notation, even though it only captures a part of the metamodel. For this purpose, EMFText allows to have separate text files next to the actual model. This enables the tool to realize a partitioning of the model fostering several developers to simultaneously work on different parts of the model, as required by Requirement 5.4 on page 92.

5.6 Movisa Modeling Workbench

Summarizing the previous sections into a dedicated modeling tool — the MOVISA modeling workbench — which complies with Requirement 5.4 on page 92 leads to a component architecture as illustrated in Figure 5.7. It is an EMF based modeling tool that was build atop the *Ecore* based core language model, providing a graphical GMF based and textual EMFText based model editor. Figure 5.7 shows that this modeling editor physically partitions a model into graphical model artifacts and textual ones. For instance, each textual model artifact, as introduced in Figure 4.5, is handled as an individual file within this modeling workbench. This partition allows to design the user interface and to configure the process data model as well as the communication relationships at the same time.

Even though the verification and the transformation engine operate on the same model, they are realized as separate components, which fosters reuse and exchangeability. The transformation engine explicitly preserves every manually written source code artifact, complying with Requirement 5.3 on page 92. Furthermore, both types of transformation rules — horizontal (ETL) and the vertical (EGL) — are separated into different packages, whereas the package containing the vertical ones additionally introduces a dedicated sub-package for each target technology. To introduce a modularity, as proposed by [Kolovos, Paige, and Polack, 2009] and required by Requirement 5.2 on page 92 and Requirement 5.4 on page 92, the code generation templates are designed using the same design strategies as in ordinary software development, such as modularization, packages, and encapsulation of routines into functions.

Using the MOVISA tool, the modeling workflow, depicted in Figure 5.8, is enforced. This mainly originates from the requirement that checking if the model

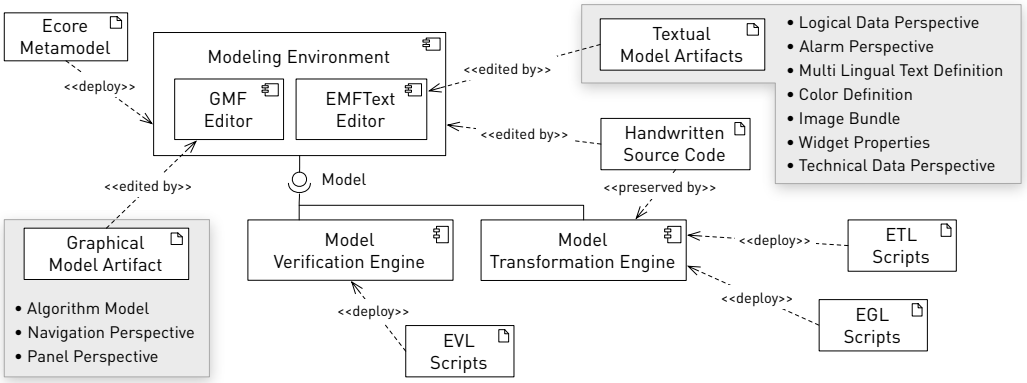


Figure 5.7: Components of the *Movisa* modeling workbench.

is wellformed should be a separate task in the modeling process (Requirement 5.1 on page 92). A positive side effect of enforcing this workflow is the vast amount of complexity which can be removed from the transformations, as model-integrity has already been proven in the previous step. Consequently, this complexity will not be removed at all. It is rather transferred into the verification rules. But unlike model transformation rules, model verification techniques have been tailored to provide an efficient tool to express model-integrity checks.

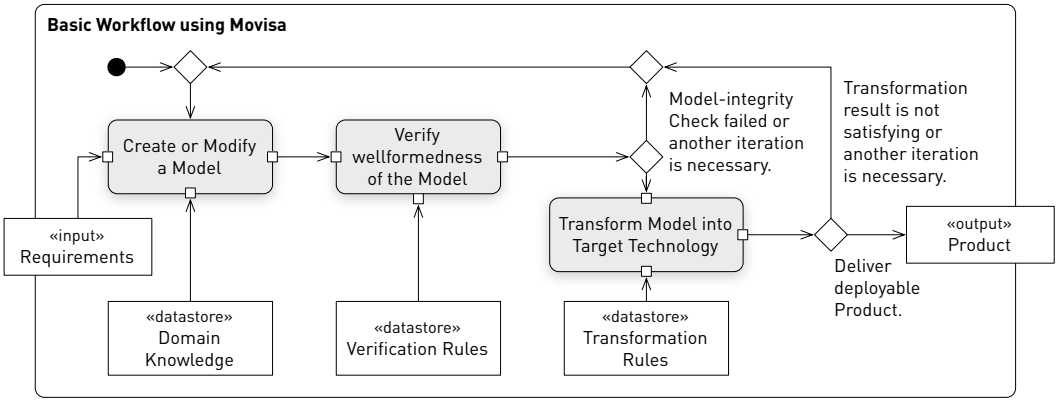


Figure 5.8: Movisa Workflow: Modeling, verifying models, and transforming models are separate tasks in the development process.

5.7 Conclusions

To conclude the previous sections, the language model of Chapter 3 and the concrete syntax notations, discussed in Chapter 4, were realized in an Eclipse based modeling workbench. It encapsulates the complexity of the language model, and it supports the modelers with a model-driven workflow (see Figure 5.8). Finally, it is distributed as easy-to-install Eclipse plugins. Meeting the requirements imposed by Section 5.1 is an important step towards meeting Requirement 2 on page 24, Requirement 4 on page 26, and Requirement 3 on page 25. Particularly the modularization of transformation rules has turned out to reduce the effort of maintaining the templates significantly, because many of these modules can be reused in templates of other target technologies.

An important result of this chapter is the interdependency between the deployment of the required components and the possibility of transforming a model into runtime artifacts for arbitrary target platforms. Thus, platform independency can only be ensured if the actual target platform is capable of providing a runtime infrastructure that, in turn, is able to execute the required components. Particularly, *sandboxed platform configurations* usually require some of these components to be transferred into dedicated processing nodes, as Section 5.3.3 points out. In conclusion, it can be stated that the more powerful the runtime environment or the target implementation technology, respectively, is, the more flexible deployment configurations can be realized.

Chapter 6

Evaluation

This chapter evaluates the findings and approaches of the previous chapters that have lead to a domain specific *Language Model* (comp. Chapter 3), its *Concrete Syntax Notation* (comp. Chapter 4), and the junction of both in a domain specific modeling workbench (see Chapter 5). First of all, an evaluation from a developer's perspective was performed by analyzing functional aspects of the MOVISA language. In this context, Requirement 1 on page 22 and Requirement 4 on page 26 will be proven by means of representative case studies. Evaluating Requirement 2 on page 24 also means to take non-functional factors into consideration. With this respect, Green (1989) provides a framework for end-user usability evaluation. However, Olsen Jr. (2007) argue that usability evaluation only works "if there is a large pool of potential users with shared expertise [...]. This does not work so well for problem domains that require substantial specialized expertise." As the MOVISA modeling workbench is made for experts of the industrial automation domain as end-users, performing an end-user usability study is not expedient with this respect. Rather, Olsen Jr. (2007) introduces a framework to evaluate user interface systems research against the background of effectiveness as convincing factor to get a new tool accepted.

After evaluating the functional aspects in Section 6.1 and the effectiveness factors in Section 6.2 based on the framework by Olsen Jr. (2007), a sound set of data is available to draw the conclusions from a developer's perspective in Section 6.3.

6.1 Functional Requirements Evaluation

Requirement 1 on page 22 determines the boundary around the application domain of process visualization solutions to monitor and operate *production automation* processes. Thus, meaningful evaluation scenarios comprise case studies of the fields *process industries*, *factory automation*, and *energy supply systems*. According to the *generality claim* presented by Olsen Jr. (2007), if a new tool is able to solve problems of as diverse domains as possible, "one can argue that the tool solves

most of the problems lying in the space between the demonstrated solutions." To raise the significance of the *generality claim*, another case study tackles to solve a problem of the *health care* domain.

In the following, the case studies are discussed individually: After introducing the particular application domain and its characteristic tasks for human operators, representative scenarios are deduced from that.

6.1.1 Case Study: Process Industries

Chemical plants like refineries, pharmaceutical facilities, and purification facilities are representatives for process industries. Lunze (2008) distinguishes between (1) *continuous* processes, implying human assignments that are characterized by continuous monitoring of process variables, such as *temperatures*, *fill levels* etc., to keep them constant or modify them in a prescribed manner; and (2) *batch* processes, which are characterized by a sequence of discontinuous subprocesses resulting in the production of the product in portions [Johannsen, 1993]. Thus, human operators are required to manage these batches.

Scenario

A laboratory plant, as a representative of a continuous chemical process, contains three tanks filled with water. Pipes connect the tanks. Pumps transfer water between them. Control valves determine the direction of the water flow. The respective process variables are provided by an OPC XML-DA service. Using a visualization solution, operators are intended to monitor a fill-level control loop of the three tanks and to manually intervene in this process by (1) parameterizing the control loops and by (2) pumping water from one tank into another one. Particularly, the following constraints must be met:

(1) A dynamically graphical user interface always showing appropriately the current values of the process variables is required. It must provide task specific mimics with relevant interaction means and must always show the relevant part of the process through dynamic graphics. Each mimic must contain an alarm component as well as a trend chart showing the evolution of the fill level process variables over time.

(2) To prevent race conditions when intervening in the process, a mutual-exclusive operation is required, meaning that only one operator is allowed to operate at the same time.

(3) To avoid situations where operators fail to release their operation token, it must be automatically released after five minutes.

(4) To keep operators informed about exceptions in the process, local alarms should be presented through an alarm control widget. In certain situations, alarms should be send to remote operators via SMS, Email, or a telephone call.
(5) The operator's actions as well as the values of the process variables representing the fill levels of the tanks and the opening width of the valves are to be logged.

Finally, it is required to deploy this visualization solution as (1) stand alone application to monitor and operate on workstations in a control room, (2) web-based solution for remote access using both laptop computers and smartphones.

Implementation of the Scenario

A good starting point to realize the requirements of the scenario is to define the required process variables and alarms. For this purpose, Listing 6.1 shows an excerpt of the Technical Data Perspective specifying an OPC XML-DA server and an item using the known OPC terminology. Listing 6.2 shows the respective excerpt of the Logical Data Perspective. It stipulates a uniform view on the data defined in Listing 6.1: Independently of the underlying communication protocol, a connection between both is drawn using the pointing to relationship.

Using the Subscription defined in Listing 6.2, it was defined that the Data Item must always be kept updated with actual process values, but only if the new value has increased/decreased by "1.0" (see Dead Band property).

Figure 6.1 presents an excerpt of the Presentation Model: It defines the required Panels (see "Process Overview Panel" as an example) and the navigation structure in between (green arrows). Each Panel contains a set of UI Components, as depicted through the images ❶ and ❷.

A distinctive feature is the Cloned Panel relationship, as introduced in Section 3.2.3 and demonstrated in the properties view ❸ of Figure 6.1: The "Process Overview Panel" defines the set of UI Components (❶) to be shown on each individual panel, namely the dynamic figure of the process (①), the required trend chart (②), an alarm control widget (③), and a row of Buttons for navigation (④). As all other Panels refer to this Panel through the Cloned Panel relationship, they inherit these UI Components and might define their own characteristic ones. The "Pump Over Panel" (❷) additionally contains the interaction means for pumping water from one tank to another one. Both red dashed rectangles (⑤) indicate where the inherited components will be inserted by the transformation.

Listing 6.1: Technical Data Perspective

```
TechnicalDataPerspective{
  OpcXmlDaServer server_1
      on http://141.30.119.241
      using de-DE provides {

    Item tp_Fuel_Level_1 {
      ItemName : Fuellstand1_Ist
      ItemPath : /Schneider
      ItemType : float
    }
}
```

Listing 6.2: Logical Data Perspective

```
LogicalDataPerspective {
  DataItem di_Fuel_Level_1 pointing to
      tp_Fuel_Level_1 {
    Type : Rational
    ItemDimension : dim_normal
    SignalOnChange : Fuel_Level_1_Changed
    Description : td_DataItem_Fuel_Level_1
    Logging : true
    MaxAge : 5000
    MinValue : 35.0
    MaxValue : 280.0

    AlarmBehavior di_Fuel_Level_1_240 {
      LimitValue : "240"
      Threshold : "5"
    }
  }

  Subscription subscription interval 1000 {
    MonitoredItem mon_Fuel_Level_1 observes
        di_Fuel_Level_1 with {
      SamplingRate : 1000
      DeadBand : 1.0
    }
  }
}
```

Listing 6.3: Alarm Perspective

```
AlarmPerspective {
  LocalAlarm a_Fuel_Level_1 {
    Priority : 0
    Description : td_Empty
    HelpText : td_Empty
    HH : di_Fuel_Level_1_240
    H : di_Fuel_Level_1_220
    L : di_Fuel_Level_1_60
    LL : di_Fuel_Level_1_40
  }

  NotificationClass
      ProcessMap_Notification {
    a_Fuel_Level_1, a_Fuel_Level_2,
        a_Fuel_Level_3
  }

  RemoteAlarm ra_Fuel_Level_1 {
    Priority : 0
    Description : td_Empty
    HelpText : td_Empty
    HH : di_Fuel_Level_1_240
    H : di_Fuel_Level_1_220
    L : di_Fuel_Level_1_60
    LL : di_Fuel_Level_1_40
    ReceiverOnHH : rar_critical
    ReceiverOnH : rar_warning
    ReceiverOnL : rar_warning
    ReceiverOnLL : rar_critical
  }

  RemoteAlarmReceiver rar_critical {
    NotificationMedia : PHONE_CALL
    Address : "+491742412729"
  }

  RemoteAlarmReceiver rar_warning {
    NotificationMedia : EMAIL
    Address : "user@company.com"
  }
}
```

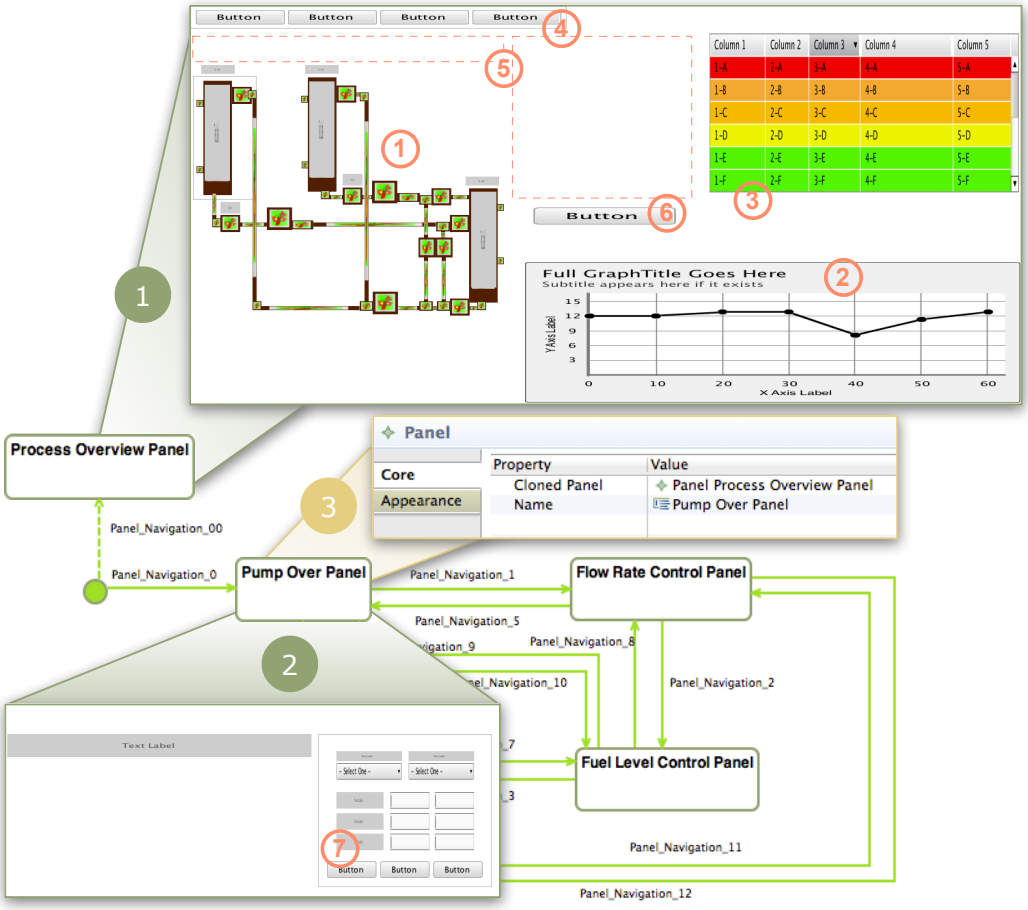


Figure 6.1: Excerpt of the Presentation Model: The *Navigation Subsystem* specifies the available Panels and the Navigation Flows between them; the *Panel Subsystem* hosts the UI Components of an individual Panel (comp. Figure 4.6). Using the Navigation Flow with the dashed arrow targeting in the "Process Overview Panel", this target Panel can be seen as an abstract one which can only be cloned by other Panels.

Specific Characteristics Implementation

VDI/VDE (1999) requires any process operation to feature at least a two-step interaction. For instance, the interaction element, being connected to the process variable, is protected by a *release* key [VDI/VDE, 2002]. It can be realized in the MOVISA tool using an additional BUTTON which sets a local DATA ITEM to "true" (see Listing 6.4). The BUTTONS to be protected contain an appropriate ANIMATION property to alter their *Accessibility* property according to the value of this local DATA ITEM (see Listing 6.5). Figure 6.2 shows a resulting faceplate after a transformation.

Listing 6.4: Setting a local DATA ITEM through a BUTTON's interaction property.

```
ButtonInteraction {
  Click Btn_Release_Click {
    Effect : Btn_Release_Effect
  }

  WriteFixedBooleanDataEffect
      Btn_Release_Effect {
    FixedBooleanValue : true
    DataItem :
        di_Pump_Over_Release
  }
}
```

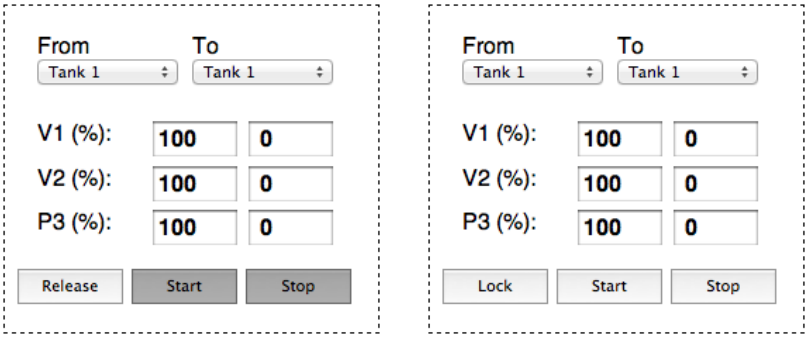
Listing 6.5: Altering a BUTTON's ACCESSIBILITY property.

```
ButtonAnimation {
  RepresentationAnimationRecord {
    TriggeringSignal :
        Pump_Over_Release_Changed

    PropertyToAnimate : State False_State {
      Visible : true
      Accessible : false
      Valid : true
    }
    PropertyToAnimate : State True_State {
      Visible : true
      Accessible : true
      Valid : true
    }

    BooleanComparator Release_Comparator {
      ReferenceValue : true
      CompareType : EQUAL
      TrueEvaluation : True_State
      FalseEvaluation : False_State
    }
  }
}
```

According to Section 3.1.2, write operations are usually required to be atomic. Section 3.4.4 specifies the semantic of WRITE DATA ITEM EFFECTS: If they are contained in a single TRIGGER, they will preferably be written as a complete set. Listing 6.6 illustrates this for the given case, because the pump over procedure requires to set (1) the source and (2) the target tank, (3) the speed of the relevant pump, and, finally, (4) the flag to start the process.

Further specific characteristics are discussed in the following.

Local and Remote Alarms. To ensure proper notifications about deviations or exceptions in the process, DATA ITEMS allow to define conditions being worth to raise an alarm through the modeling entity ALARM BEHAVIOR, as shown in



(a) BUTTONS are locked for interaction.

(b) Interaction lock has been released for process operation.

Figure 6.2: Faceplate with a two-stage operation: Using the BUTTONS "Start" and "Stop" is only possible after actuating the "Release" BUTTON.

Listing 6.2. LOCAL and REMOTE ALARMS, specified in the ALARM PERSPECTIVE (see Listing 6.3), aggregate ALARM BEHAVIORS to give them a particular meaning. In this case, a LOCAL ALARM is referring to this ALARM BEHAVIOR via the HH relationship and, hence, it causes a *High-High* alarm. The ALARM CONTROL widget, shown in Figure 6.1 (③), presents this alarm to operators, as it refers to the NOTIFICATION CLASS embracing this alarm (comp. Listing 6.3) and uses its characterizing ANIMATION PROPERTY, as shown in Listing 6.7.

Listing 6.6: BUTTON INTERACTION for realizing an *atomic write* operation.

```
ButtonInteraction {
  Click Pump_Over_Btn_Start_Click {
    Effect : Select_From_Effect
    Effect : Select_To_Effect
    Effect : Input_P3_Effect
    Effect : Start_Effect
  }
  WriteEnteredDataEffect Select_From_Effect {
    ValueContainingWidget : Select_From
    DataItem : di_Tank_A
  }
  WriteEnteredDataEffect Select_To_Effect {
    ValueContainingWidget : Select_To
    DataItem : di_Tank_B
  }
  WriteEnteredDataEffect Input_P3_Effect {
    ValueContainingWidget : Input_P3
    DataItem : di_P3_FL
  }
  WriteFixedBooleanDataEffect Start_Effect {
    FixedBooleanValue : true
    DataItem : di_Start_Pump_Over
  }
}
```

Listing 6.7: Defining the alarms to be presented in a particular ALARM CONTROL widget.

```
AlarmControlAnimation {
  NotificationClass :
      ProcessMap_Notification
}
```

For notifying off-site operators, Remote Alarms are used very much like Local Alarms. However, they additionally allow to configure which Remote Alarm Receiver should be notified in certain situations, as depicted in Listing 6.3.

Due to the requirement of supporting web-based solutions, monitoring the alarm conditions and notifying operators must take place on a centralized host. The transformation creates the necessary configuration files according to the information of the Alarm Perspective.

Ensuring Mutual Exclusive Operations. Operating technical processes mutual exclusively was not intended by the language per se, but it can be realized using the Algorithm Model. First, a centralized managed token is necessary for which several clients are competing. It is realized as a process variable on the OPC XML-DA server. A "false" value indicates that no client is operating at the moment and, hence, it can be obtained using the Button ⑥ in Figure 6.1. Actuating this button establishes the State "OTM_Request_Token" in Figure 6.3. Consequently, its containing Actions indicate that the operation token has been obtained. For this purpose, a local Data Item is used. Based on it, the faceplate, shown in Figure 6.2, becomes accessible. Thereafter, the centralized token is set to "true". This causes the Button ⑥ to become inaccessible for other visualization clients.

To hand over the operation token after accomplishing the desired task, the State "OTM_Release_Token" resets both the local and the centralized token.

Timed Triggering of Scripts. Requesting the operation token entails a user triggered *script* activation, but it should be automatically returned after five minutes. As *time events* were excluded by OMG (2008, p. 44), because they always imply platform specific concerns, the required timer will be realized using the Boundary mechanism. Figure 6.3 presents its two interfaces. The Required Interface is sensible to the Signal "Release_Timer_Start", which is sent by the Send Signal Action "Start Timer". Listing 6.8 shows the code (placed inside the protected region) to be executed after receiving this Signal: It starts a timer and sends another signal after five minutes. This causes the State "OTM_Release_Token" (see Figure 6.3) to be established and, thus, to hand over the operation token.

Logging. As the required transformation targets are also comprised of web-based solutions, preserving the history of process variables has to take place on a centralized host which is able to run a database server. The transformation creates

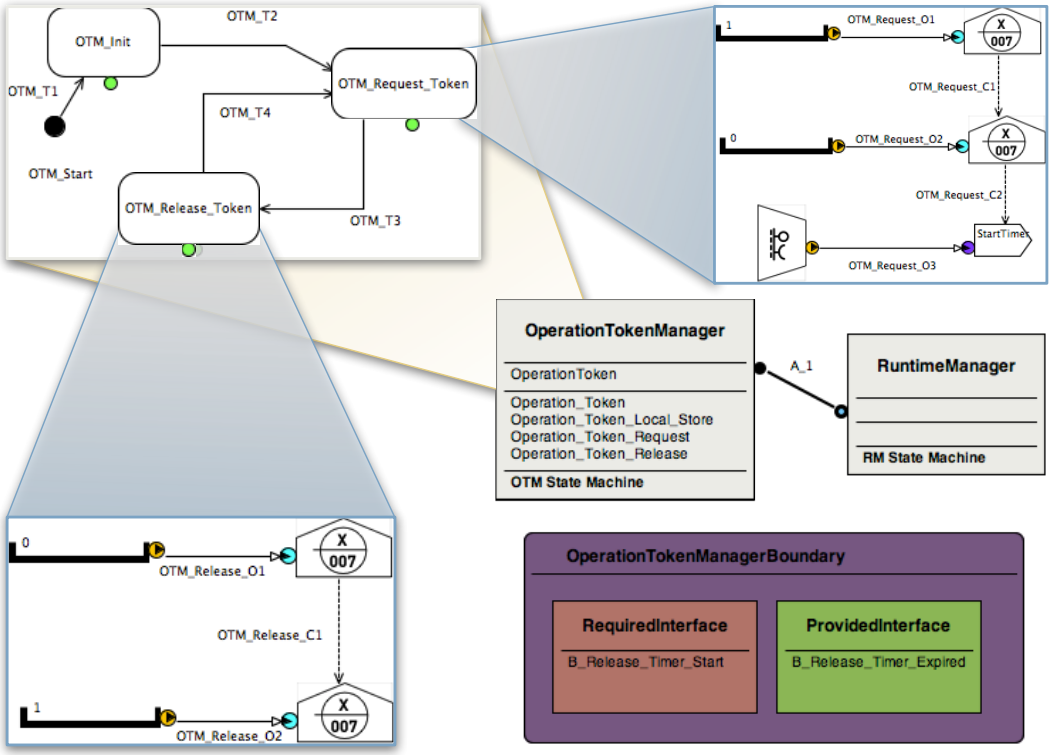


Figure 6.3: Excerpt of the ALGORITHM MODEL ensuring to operate the technical process mutual exclusively using Executable UML.

Listing 6.8: Platform specific BOUNDARY realization ensuring to send a SIGNAL after a certain time slot expired.

```
if (event instanceof Release_Timer_Start) {
  /* protected region PreserveRelease_Timer_Start on begin */
  // Put here the code for 'Release_Timer_Start'
  window.setTimeout(function() {
    eventBus.fireEvent(new Release_Time_Exceeded(), null);
  }, 300000);
  /* protected region PreserveRelease_Timer_Start end */
}
```

the necessary SQL[1] scripts to populate a relational database with the required table structure. Additionally, the transformation creates a database server configuration file considering all Data Items of the model with the Logging attribute set to "true".

Code Generation for Different Platforms. Basically, the visualization is to be executed on two different hardware platforms: (1) on a workstation in a control room and a laptop for remote operation, and (2) on a mobile device. For this purpose, Figure 6.4 shows an excerpt of the Presentation Model. The tank and the fill level sensors shown in this figure are surrounded by a Simple Container (❶). They are always shown as a whole (comp. Definition 3.1 on page 51). Furthermore, they are annotated to be kept on big and medium-sized screens. On small screens, they will be replaced by a Text Label component. On this occasion, the CUI-to-CUI transformation converts the height animation of the rectangle representing the fill level to a Value Output Animation and adds it to the Text Label. The Image (❷), representing a pipe, is annotated to be removed by the CUI-to-CUI transformation for medium and small screens. This transformation, however, translates the visualization solution into another context of use only by means of removing and converting UI Components. Thus, a manual post-treatment is necessary to relayout these components.

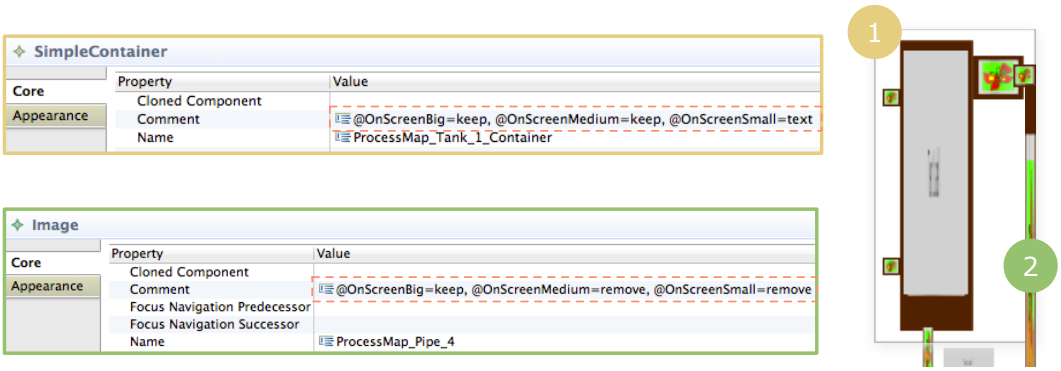


Figure 6.4: Excerpt of the Presentation Model emphasizing on the model annotations as instructions for the *CUI-to-CUI* transformation: The Simple Container (❶) component will preserve its contents on big and medium sized screens and will be converted to a Text Label on small screens. The Image (❷) will only be kept on big screens.

[1]The *Structured Query Language* (SQL) is a language to express queries on the datasets of relational database systems.

Results

A user interface model was developed in the previous sections. This manually created model using the MOVISA modeling workbench, was designed for hardware platforms with big-sized screens. Due to the requirement to provide a mobile visualization solution, a respectively tailored second model was generated based on transformation instructions in form of model annotations. Subsequently, *CUI-to-FUI* transformations create the following runtime artifacts:

(1) *Python* based non-sandboxed visualization solution (see Figure C.1 and Figure C.2) to be operated on the workstation host in the control room,
(2) *HTML* web-based sandboxed visualization solution (see Figure C.3 and Figure C.4) to be run within a web browser of a laptop computer for remote operation,
(3) *HTML (mobile)* web-based sandboxed visualization solution (see Figure C.5) to be executed within a mobile webbrowser of a smartphone.

The runtime artifacts for solution (1) and (2) were derived from the first model and the artifacts for solution (3) from the second one. Figure 6.5 presents all generated artifacts together with their particular deployment targets.

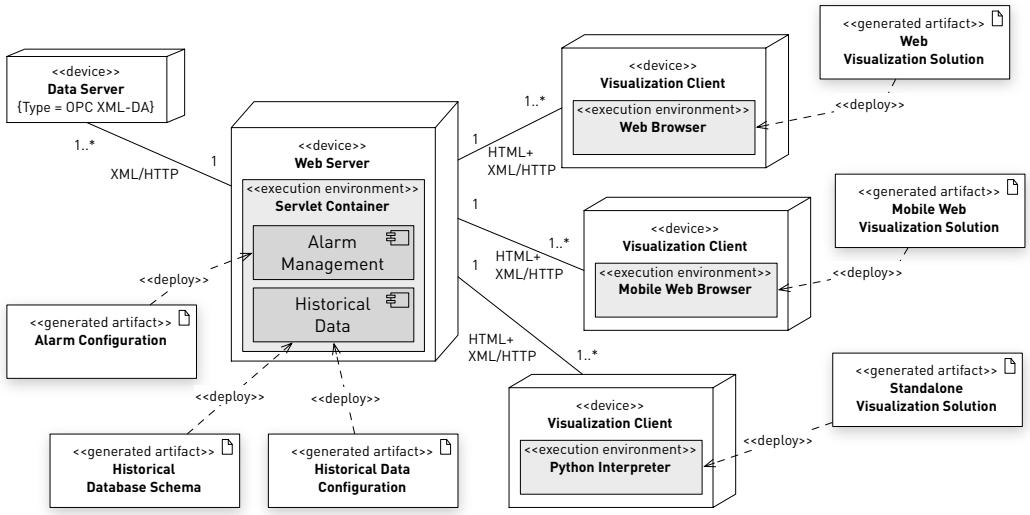


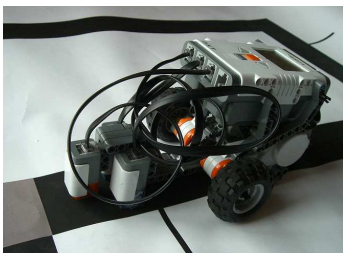
Figure 6.5: Generated artifacts in relation to the resulting deployment configuration of the process industries case study: A centralized web server operates the *Alarm Management* and *Historical Data* components, different visualization clients are equipped with completely generated visualization solutions of different target technologies.

6.1.2 Case Study: Autonomous Robots in Factory Automation

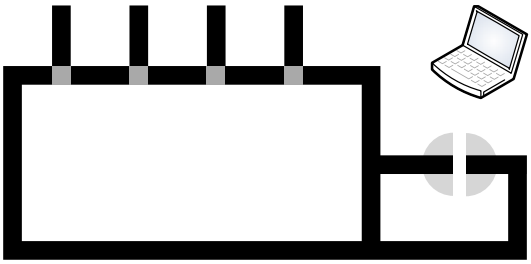
Technical processes that use machine tools or robots to manufacture piece goods — e.g. through grinding or milling — belong to the field of factory automation that is characterized by discrete processes. Usually, machining happens at production and assembly lines, but also autonomous vehicles are used to transport the workpieces from one workstation to another one. Human operator assignments are among others the planning and supervising of the production orders [Johannsen, 1993].

Scenario

Lego Mindstorms™ NXT [Lego, 2011] robots serve as ideal models of autonomous vehicles. Two of those robots, configured as shown in Figure 6.6a, are moving on a production line model, as depicted in Figure 6.6b, featuring the dimensions of $2\,m \times 1\,m$. This production line is comprised of one loading station and four workstations. The robots use light sensors to follow the black path, identified using RFID techniques, from the loading station to the workstations, where they symbolically wait until the production order is accomplished. Then they drive back to the loading station and wait for the next job. A centralized control station computer (see Figure 6.6b) holds a *Bluetooth* connection to each NXT robot and ensures the distribution of the production orders as well as a collision-free operation.



(a) Setup of the NXT vehicles: They are equipped with light and RFID sensors.



(b) Production line model with one loading station, four workstations, and a centralized control station.

Figure 6.6: Setup of the factory automation case study.

Human operators are in charge of monitoring this production process. They must always know the robot's position on the production line and which production order they are currently carrying out. They operate this process by composing a queue of production orders or by deliberately supplying individual vehicles with particular orders.

Implementation of the Scenario

Creating the user interface for this scenario is very similar to the procedure in Section 6.1.1. The production line can be shown as static Images and each robot as a dynamic Image, animating its Position properties, on top. Furthermore, user interface elements with standard interaction means for managing process orders are required. More importantly, however, is the realization of the connection to the process. The centralized control station provides a proprietary and still not supported web service based interface to its process data. Thus, there are three variants for integration:

(1) Using the Boundary to implement the necessary communication routines within the generated code;
(2) employing the Generic Server, as discussed in "Technical Data Perspective" of Section 3.2.2, and, thus, enhancing the transformation rules; or
(3) enhancing the language model to support another server.

Variant (1) is advisable if a particular data server specification is only used once. Otherwise, variants (2) or (3) should be taken into consideration. One has to balance a more easy integration using the Generic Server approach against the possibility to conveniently create the models. Here, the terminology provided by the particular data service specification is used when it was integrated into the language. In this particular case, the control station web service is the basis for further projects in an educational environment. Students should be able to easily visualize their individual NXT projects. Hence, it is reasonable to enhance the language model with the new server. The following steps must be performed:

(1) Enhancing the *Core Language Model* with the new Server element Ifa Nxt Control Ws Server and the respective Server Data Item element, as depicted in Figure B.64 and Figure B.70,
(2) realizing new *Language Constraints*,
(3) creating a *Concrete Syntax Notation* for the new elements, as exemplary shown in Listing 6.9,
(4) providing suitable transformation rules, whereas generalized routines of the existing transformations can be used (e.g. the routines to assemble SOAP messages), and
(5) integrating these aspects into the Movisa tool environment.

Listing 6.9 shows that a single item is always comprised of the NXT bluetooth address Nxt Address and of a command Nxt Command to be executed on the control station. Using the first item (line 5 — 8), the control station returns the production order which is currently carried out by the robot, identified with the

Listing 6.9: Excerpt of the TECHNICAL DATA PERSPECTIVE showing the configuration of the newly integrated data provider.

```
1  TechnicalDataPerspective{
2    IfaNxtControlWsServer nxt_controller on
3        http://localhost:8080/axis/services/IfaNxtControlWs provides {
4
5      Item Server_NXT8_ActualOrder {
6        NxtAddress : 00165308F1FC
7        NxtCommand : ActualOrder
8      }
9
10     Item Server_NXT8_ActualPositionX {
11       NxtAddress : 00165308F1FC
12       NxtCommand : ActualPositionX
13     }
14   }
15 }
```

given address. However, as the respective data structure with which the order is encoded is a byte string, it has to be decoded through suitable routines that were encapsulated behind a BOUNDARY element.

Results

Due to the not yet supported data server, the implementation of this case study resulted in a full integration of this data server into the MOVISA tool. Based on this, the modeling endeavor created a MOVISA model, which captured the operative characteristics of a visualization solution and was intended to monitor and operate a factory automation scenario. Subsequently, a transformation produces the runtime solutions shown in Figure C.6 (Python as non-sandboxed solution) and Figure C.7 (HTML as sandboxed solution to be executed by a webbrowser). Figure 6.7 presents all generated artifacts together with their particular deployment targets. To conclude, independently of the concrete runtime technology, the user interface contains means of interaction to distribute production orders to the NXT vehicles and means to monitor their respective states.

6.1.3 Case Study: Energy Supply Systems

Lunze (2008) characterizes energy supply systems by centralized as well as decentralized automation facilities dedicated to ensure that distributed energy producers reliably supply consumers with electrical energy, heat, or gas. Operators need a wide range of information to keep the process variables constant and the supply network trouble-free as well as to plan the operating times of energy producers [Lunze, 2008]. Depending on the network to be monitored and operated, the field of *energy supply systems automation* is close to the field of *building automation*, which ensures,

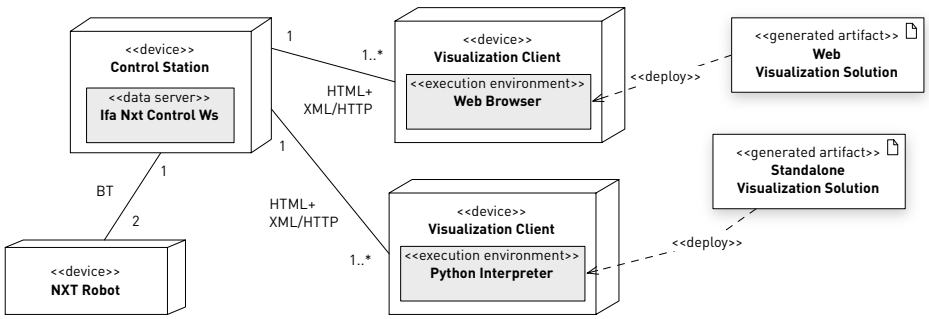


Figure 6.7: Resulting deployment configuration of the factory automation case study: The control station "Ifa Nxt Control Ws" controls the production process and provides a web service based interface for visualization clients. Two different runtime solutions were generated from a single MOVISA model.

among other things, optimal climate conditions in buildings, trouble-free operation of IT-infrastructure networks, and reliable power supply.

Scenario

Large building complexes like the campus of the *Technische Universität Dresden* need constant monitoring of its supply networks. Particularly, a trouble-free electricity supply must be ensured due to laboratories and similar facilities. Figure A.1 and Figure A.2 show an excerpt of the existing visualization solution to supervise the power supply of these buildings. It is characterized by a hierarchical structure starting at an overview display of the entire university and ending at several rooms of one building. Operators are able to navigate through this structure to capture current process values and system parameters. An alarm management system notifies the operator in case of too high temperatures of transformers or if the power network is getting overloaded. A very specific characteristic is the use of *conditional three-stage switches* (comp. "S01" and "S10" of "Trafo 1" in Figure A.1): The first stage is locally operated by a hardware switch which indicates whether an operation is allowed by the visualization solution. The following two stages are represented by a switch being protected with a separate *release* key. Operating switch "S01", however, is associated with the condition that "S10" must be in the state "off".

Furthermore, in this case study's scenario, a decentralized access to the visualization solution is required so that the facility managers of each building are able to monitor the current states of their area of responsibility.

Implementation of the Scenario

Composing the user interface and particularly the Presentation Model part to realize the current scenario is very similar to the implementations carried out in the previous case studies: Image components are used to constitute the process devices — the electrical components and the wires in between — on the screen. The operating states of these devices are animated by discrete Image Animations. Having this in mind, the *three-stage* switches can be realized as follows: An appropriate process variable is connected to the local switch. It indicates whether its position is "on" or "off". If it is in state "on", the release key gets operational what is ensured by a Representation Animation. Hence, realizing this requirement is very much like the approach of meeting the mutual exclusive operation requirement discussed in Section 6.1.1. The possibility of actuating switch "S01" depends on the process variable being connected to the switch "S10". Meeting such requirements has also be shown in "Implementation of the Scenario" of Section 6.1.1.

Another aspect to be considered while pursuing this case study is to supply real process data. For this reason, a simple network simulator *NetSimP*, based on *State Chart XML* [Barnett et al., 2011], was developed and equipped with a web service interface. Each state represents the complete data model of the electricity network in a distinct point in time. Thus, it provides an independent snapshot of the entire network. Timing conditions and events annotated to the transitions enable automatic as well as user triggered state changes. Figure 6.8 demonstrates the fundamentals of *NetSimP* using a trivial example.

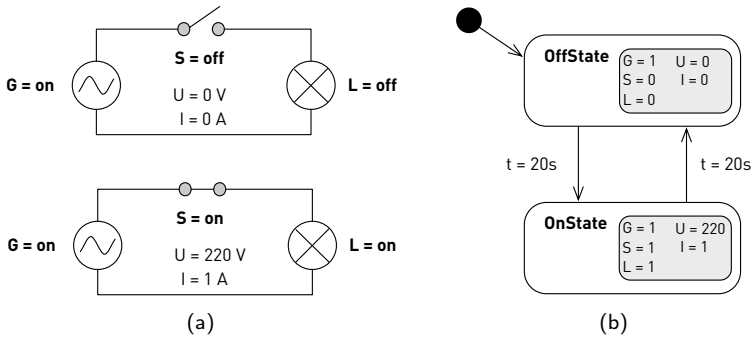


Figure 6.8: Demonstration of the basic principle behind the network simulator *NetSimP*: Each state represents the data model of the entire network in a distinct point in time.

To use the process data, provided by the *NetSimP* service, in a Movisa model,

this still not supported web service needs a terminology of its own in the Technical Data Perspective of the Movisa language. Based on the discussion in "Implementation of the Scenario" of Section 6.1.2, the concretization of a Generic Server element reflects a good compromise between reuse and integration effort. Neither does the metamodel nor the concrete syntax notation need to be modified or enhanced. Only existing *Language Constraints* are to be expanded accordingly and a new set of transformation rules is to be provided.

As an illustration, Listing 6.10 presents an excerpt of the configuration of the example network demonstrated in Figure 6.8 expressed using the terminology of the Generic Server. The only language constraint is that an Item of the Generic Server of type "NetSimP" must contain exactly one Generic Parameter "Process Variable".

Listing 6.10: Excerpt of the Technical Data Perspective showing a configuration of the sample network presented in Figure 6.8 through the Generic Server terminology.

```
TechnicalDataPerspective {
  GenericServer SampleNetwork of type NetSimP on http://141.30.119.241 provides {

    Item tp_S {
      GenericParameter ProcessVariable : "S"
    }

    Item tp_G {
      GenericParameter ProcessVariable : "G"
    }
  }
}
```

Results

The main outcome of this case study is that the Movisa modeling workbench is suitable to create a visualization solution model for a real scenario. Even though it does not use real process data and the model only focuses on a few sections of the university's building complex, it could be proven that the set of building blocks of the modeling language is sufficient to capture the operative characteristics of a visualization solution to monitor and operate an energy supply system. Section C.3 shows the generated runtime solutions, whereas Figure 6.9 depicts the concrete deployment of the generated artifacts.

6.1.4 Case Study: Health Care

Systems for monitoring the vital functions of humans in hospitals are very similar to systems for supervising technical processes. These very specialized systems

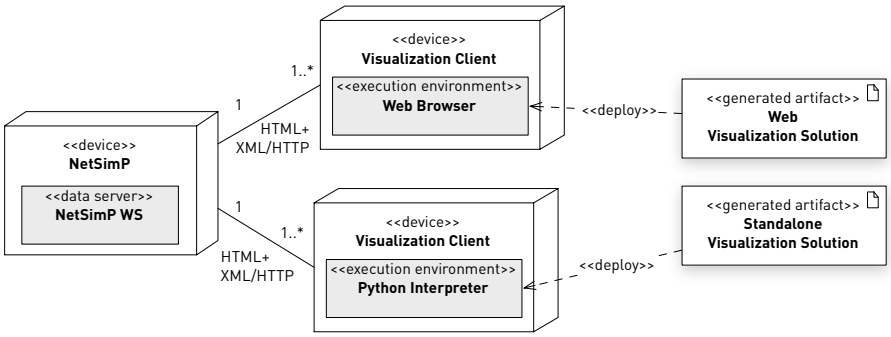


Figure 6.9: Resulting deployment configuration of the power supply system case study: Runtime solutions of two different target technologies were generated. Both are connected to the network simulator *NetSimP*.

gather data from sensors. Based on this, they either allow the hospital personnel to parameterize the life support system or the process is done automatically. Another, broader sector comprises devices that are used by patients during therapies: as information system or to keep records of patient's vital functions to be sent to the attending physician. Consequently, there are high demands on data security.

Scenario

Diabetes patients need intensive care from their attending physicians. To carry out long term treatments, patients regularly have to provide relevant information in a comparable fashion about their meals, their sports activities, and about other medicine they have consumed. Additionally, the patients have to check their blood sugar level and also have to provide this information. For patients' convenience, they are supported by an application dedicated to gather the data in form of surveys. After filling in a survey, the application stores the particular data records on a server to which the attending physician has access. Two important constraints must be met:

(1) The application is required to encrypt this very personal and, thus, sensible information using a *Public-Key* procedure. Only the attending physician is authorized to read it.

(2) It must be ensured that the patients completely fill in the forms and send complete data sets to the physician.

Implementation of the Scenario

Unlike the visualization solutions carried out in the previous case studies, this application does neither need to monitor nor to operate a technical process. Only the transmission of an encrypted data set is necessary. However, in the current development state, MOVISA does not include security measures other than a low level encryption of the communication media. Given this fact and the very specific nature of the data receiver, realizing this functionality involves the development of appropriate routines behind the BOUNDARY. Consequently, the definition of a CLIENT DATA MODEL is not necessary at all. Users enter the required data through standard forms based on INPUT fields, CHECK BOXES, and RADIO BUTTONS. Two alternatives exist for processing and storing this data:

(1) A set of actions, triggered by a user interaction, collects the entered data (READ PRESENTATION MODEL ACTION) and stores it in a dedicated object. During this collection process, the entered data is checked for completeness and validity. Thereafter, it is either sent to the boundary for encryption or the user is asked to provide the outstanding data.

(2) Instead of realizing the procedure mentioned above using appropriate ACTIONS of the ALGORITHM MODEL, it can completely be expressed through platform specific source code behind a BOUNDARY.

Even though pursuing the first variant entails more efforts, it has the advantage of capturing the functionality at a pure functional and, thus, platform independent level. Deploying such an application to another runtime configuration is subject to a transformation. The second variant, in contrast, requires to redevelop these functions. Thus, it is recommended to minimize the use of the BOUNDARY to functional aspects that cannot be expressed using the existing language constructs.

The functionality to load the necessary public key and the implementation of the cryptographic routines are to be expressed using platform specific source code behind a BOUNDARY. Accessing data on a computer's file system is out of scope of (F)UML, as it is a very platform specific subject matter. The effort for reinventing cryptographic algorithms using the action notation instead of employing operating system specific *crypto libraries* justifies the benefits by no means.

Results

Except the cryptographic function, all functions were realized using the MOVISA language and stored in a model. Transformations, however, could only be performed to generate a *non-sandboxed* target technology, as file system access is necessary to load a public key. Web-based solutions, on the other side, are not allowed

to have access to the clients file system. Furthermore, web-based solutions — and particularly mobile solutions — might not be powerful enough to calculate cryptographic algorithms. Section C.4 shows the resulting visualization solution based on the target programming language *Python*. It meets all requirements of the given scenario. Figure 6.10 presents the deployment of the runtime solution.

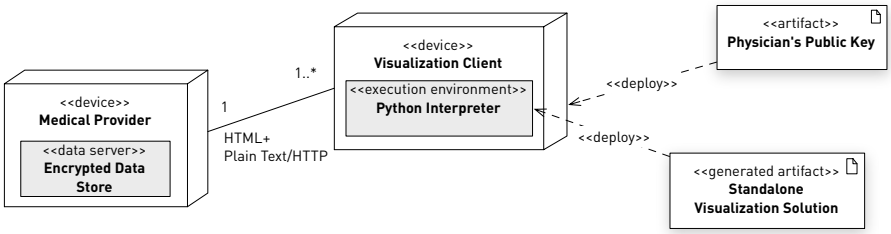


Figure 6.10: Resulting deployment configuration of the health care case study: Only a *non-sandboxed* runtime solution is able to meet the requirements.

6.2 Effectiveness Factors Evaluation

In the previous section, the MOVISA language was proven to be capable of sustainably capturing operative characteristics of visualization solutions and based on this, these solutions can be deployed to different runtime configurations. In this section, the MOVISA language will further be analyzed with respect to its effectivity. For this purpose, the evaluation framework of [Olsen Jr. 2007] is used, which attests a tool to be effective if it complies with the following claims: (1) generality, (2) reduces solution viscosity, (3) addressing a new problem, and (4) power in combination. While meeting the *generality* claim has already been proven by providing solutions to problems of different domains by means of the case studies, discussed in Section 6.1, the other claims will be discussed in detail in this section.

6.2.1 Reduce Solution Viscosity

Good tools foster, according to [Olsen Jr. 2007], the efficient creation solutions by reducing viscosity in the development process. In the context of this thesis, it means that domain experts are supported in focussing on expressing their domain knowledge to provide solutions to given problems. Olsen Jr. (2007) states that there are at least three ways to reduce solution viscosity: flexibility, expressive leverage, and expressive match.

Flexibility

According to Olsen Jr. (2007), a tool is *flexible* if it allows developers to make rapid design changes. MOVISA complies with this claim in various ways: (1) As the MOVISA tool contains transformations to create runtime artifacts for different target platforms, design changes in a MOVISA model can simply be rolled out to existing solutions. In the reverse case, it is well-known that technology evolution is closely related to changes in software stacks and hardware interfaces. Using a model-driven approach and particularly the MOVISA tool, reacting to such changes only requires to adopt the transformations and to apply them to existing models. (2) The MOVISA language model provides the CLONED PANEL or CLONED COMPONENT references, respectively, fostering the reusage of UI COMPONENTS or their properties. Changes in these components or properties are automatically applied to the cloned ones. (3) Due to the separation of the CLIENT DATA MODEL into LOGICAL and TECHNICAL DATA PERSPECTIVE a concrete data server specification can simply be replaced by another one while keeping the logical data model and, therewith, the PRESENTATION MODEL and the ALGORITHM MODEL unchanged. It is then up to the transformation to choose an optimal integration strategy which meets the logical data configuration.

Expressive Leverage

When a tool reduces the number of possible design choices that a modeler must make to express a particular solution, Olsen Jr. (2007) refers to the characteristics of *expressive leverage*. Due to the model driven nature of the MOVISA language, modelers are enabled to accomplish more by expressing less, because only functional aspects must be defined. For instance, DATA ITEMS only need the attribute LOGGING set to "true" and the transformation automatically creates the required artifacts to set up a database to record the values of the associated process variables.

Expressive leverage is additionally achieved by eliminating repetitive design choices [Olsen Jr. 2007]. For this reason, MOVISA features the following means: (1) Modelers are supported by means of libraries of preconfigured modeling elements — the COMPLEX UI COMPONENT plays a key role in this respect. (2) The CLONED PANEL and CLONED COMPONENT references were introduced to reuse components and their properties. (3) Deploying a MOVISA model to another target technology only requires another transformation. (4) Deploying a MOVISA model to another hardware platform possibly entails a prior CUI-to-CUI transformation. For this purpose, only annotating the model is required to instruct the transformation how to deal with particular UI COMPONENTS. Even though some post treatment

is necessary, the transformation is able to create many aspects automatically, e.g. the Animation Properties if UI Components were replaced. In the best case, only a rearranging of the UI Components is required.

Expressive Match

Expressive match is a measure of how close the means to express solutions are to the problem being solved [Olsen Jr. 2007]. In the context of a domain specific language, this claim mainly addresses the concrete syntax notation. As Movisa provides graphical and textual syntax notations, depending on the aspect of a visualization solution to be expressed, it can be argued that this language is a very close match to the design problem. Thus, it provides a good cognitive fit to the domain expert's thinking model: (1) The Presentation Model allows to determine the navigation hierarchies using a graphical notation that is inspired by a state machine. While the *Panel Perspective* enables to design the user interface with a visual feedback, properties of the UI Components can be expressed in an efficient text based notation. (2) The Algorithm Model constitutes a scripting language that is graphically expressed using the same notations as if discussing on whiteboards. (3) The Client Data Model is used to express a vast amount of possible data. For this purpose, a lightweight and efficient text-based concrete syntax notation has been provided which is very similar to the means of expressing data structures with general purpose programming languages.

6.2.2 Addressing a New Problem

If a new tool is able to provide solutions to problems which existing tools are not capable of, Olsen Jr. (2007) attests this new tool a great demand for attention. Unlike existing tools (e.g. *WinCC* or *InTouch*, comp. Section 2.1), which need dedicated runtime environments to execute a visualization solution, the Movisa tool is able to deploy models of visualization solutions to different platforms and runtime configurations.

6.2.3 Power in Combination

Effective tools enable to combine their provided, probably more basic, building blocks to create solutions of any complexity. According to Olsen Jr. (2007), a fundamental prerequisite is the availability of clearly defined mechanisms for combination. In this way, Movisa defines a limited set of Elementary UI Components that are able to be combined to more complex pieces using the Complex UI

Component. These complex components specify simple interconnection interfaces (comp. Figure 3.12) based on Signals. Consequently, instead of implementing the interface of all existing complex components, a new element must only be compliant to this standard interface. Moreover, using the Algorithm Model and particularly the Boundary concept, arbitrary new functionality can be integrated into the language and it becomes accessible from all existing components through the Signal interface. Even though this mechanism is very simple and straightforward, it is very powerful.

6.3 Conclusions

This chapter evaluated functional aspects of the domain specific modeling language Movisa by means of representative case studies and non-functional aspects according to the evaluation framework proposed by Olsen Jr. (2007). Basically, the case studies demonstrated that Movisa is capable of sufficiently expressing the operative characteristics of visualization solutions to supervise technical processes in industrial automation: Using the standard building blocks of the Movisa language, the user interfaces on which the individual scenario insisted could sufficiently be expressed. Based on modeling process data and communication relationships, the user interface components are able to reflect the operative process states and allow interventions — Movisa complies with essential representation, animation and interaction properties. Even though complex user interface components can be composed, Movisa currently lacks techniques for separating these preconfigured components from the individual model into a centralized repository. However, as these are coherent presentation units, they form a key role when deploying visualization solutions to different hardware platforms, which has also been shown in Section 6.1.1. Furthermore, Section 6.1.1 proves that automation specific concerns such as atomic write operations, logging process data, and providing an alarm management system can be expressed through the Movisa language as well.

If a scenario requires specific functionality that the Movisa language does not provide standard building blocks for, there are different extension points: (1) The Algorithm Model allows to express custom functionality with either user or timed triggered actuation. Together with (2) the Boundary, it also allows to realize very platform specific concerns, e.g. to interpret raw sensor values, as shown in Section 6.1.2. However, the more platform specific this functionality is, the less diverse runtime artifacts can be generated. For instance, Section 6.1.4 demonstrates that realizing cryptography algorithms excludes certain platforms as possible runtime targets. Even though the Boundary can be used to integrate

a new data server specification (comp. Section 6.1.4), MOVISA provides (3) the GENERIC SERVER modeling entity to reuse new data server specifications in future projects. While realizing the *energy supply system* case study in Section 6.1.3, such kind of data server was defined and new transformations worked out. The modular nature of the transformations fosters an easy integration. To unfold full modeling efficiency, though, (4) the MOVISA language model has to be enhanced. This could be demonstrated through completely integrating a new data server specification in Section 6.1.2.

Evaluating effectiveness factors show that also non-functional aspects could successfully be evaluated: (1) The MOVISA modeling workbench encapsulates the complexity of the language model behind a high-fidelity concrete syntax notation allowing an efficient modeling endeavor. (2) The verification tool allows to identify errors in early design stages. (3) The transformation enables automatic runtime artifact generation, whereas the extensible nature of transformations fosters maintainability.

Finally, performing the case studies shows that MOVISA is capable of sustainably describing visualization solutions for the application domain *production automation* and of generating respective runtime solutions. Limitations of the MOVISA approach could be identified while creating a solution within the domain *health care*. However, these limitations are mainly related to different concepts of handling "process" data. Despite these domain specific characteristics, the requirements of the respective scenario could be met by using extension points of the language model, namely the BOUNDARY. Consequently, the significance of the *generality claim*, as proposed by Olsen Jr. (2007), could be raised. To conclude, the identified potential of MOVISA as well as the mentioned limitations strengthen the statements of authors like Fowler (2011); Kelly and Pohjonen (2009); Stahl et al. (2007); Völter (2009): Domain specific languages should always be created with an explicit use case or application domain in mind. They unfold their full potential for this limited target domain. However, providing reasonable extension points can broaden the range of application.

Chapter 7

Exploiting Movisa in Supplementary Model-Based Environments

Getting a new tool, a method, or a language accepted in industry demands that this new approach integrates in and cooperates with other engineering or development procedures. Figure 2.1 shows that HMI development in process industries occurs very late in the overall plant engineering process. As the majority of *Computer Aided Engineering* (CAE) data was already been made available, the *autoHMI* [Urbas and Doherr, 2011] approach tries to derive concrete UI designs from this data. When it comes to also considering usability attributes, the end users — the operators — need to participate in the design process. For this purpose, two engineering processes were established: (1) the *Useware Engineering* [Zühlke, 2004] for the development of user interfaces for technical systems and, more generally, (2) the *User Centered Design* (UCD) [Norman and Draper, 1986] process.

This chapter presents three endeavors to integrate the MOVISA language into these established engineering procedures. Namely, the autoHMI integration is introduced in Section 7.1 to address Requirement 5 on page 27. The integration into the Useware engineering process is shown in Section 7.2, and the integration of MOVISA into the UCD process is demonstrated in Section 7.3 to address Requirement 6 on page 28. Finally, Section 7.4 draws the conclusions.

7.1 autoHMI: HMI Generation in Process Industries

According to Urbas and Doherr (2011), visualization solutions to monitor and operate technical processes are built at the very end of the overall engineering procedure, as discussed in Section 2.1 and, particularly, depicted in Figure 2.1. The reason for this is that required data to design the user interface only gets

available during the engineering process of the plant and many interdependencies exist between the automation structure and the HMI design. On the other side, there is a high demand (1) to also incorporate operators into the early stages of design processes and (2) to reuse the engineering data in the HMI development phase, as stressed by Requirement 5 on page 27.

Urbas and Doherr (2011) propose *autoHMI*, a model driven tool chain enabling to start with generating the HMI at an early engineering stage. For this purpose, the autoHMI concept is based on several abstract user interface models that can simply be mapped onto the CAMELEON REFERENCE FRAMEWORK. These models are populated with data, which are available at the relevant point in the engineering timeline. Figure 7.1 presents the basic concept of autoHMI.

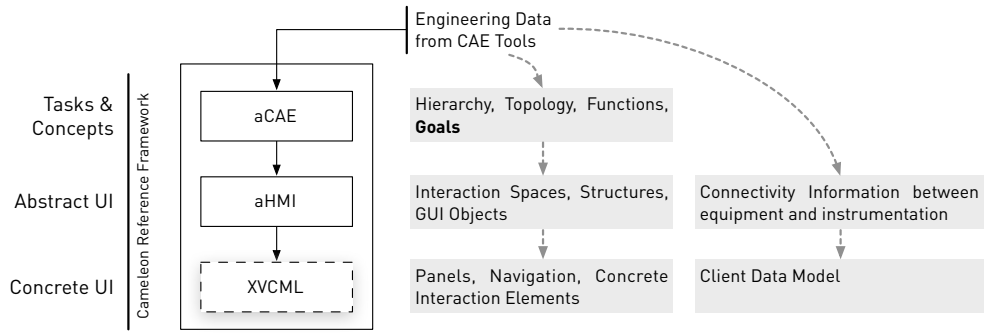


Figure 7.1: Classification of the *autoHMI* concept (white box in the center) by means of the CAMELEON REFERENCE FRAMEWORK (left hand side). The gray boxes present the individual information gathered from the engineering data and stored in the particular models.

Based on the CAE data, derived directly from engineering databases, the *aCAE* model includes, among other things, information about the placement of units, equipment, and control modules. Even though this information cannot directly be used to generate an HMI, it provides important information about hierarchies and topology. The *aHMI* model enriches this information by mapping it to a hierarchy of interaction and navigation spaces. This forms the basis for the *Concrete User Interface* model that, in [Urbas et al., 2011], the *XVCML*[1] language [Braune, Hennig, and Schaft, 2007] was used for. Based on the information stored in the aHMI model, an XVCML model defines the concrete interaction elements. Hence,

[1] XVCML is the XML-based and far more simple predecessor of MOVISA. All its concepts and lessons learned were moved into the *Eclipse* ecosystem and were further developed under the name MOVISA.

it determines the graphical screen based user interface.

In a later point in time, when the actual equipment — devices and data protocols — has been determined, aHMI is populated with connectivity information between equipment and instrumentation. Then, this information can be obtained from engineering databases as well. Based on this kind of information stored in the aHMI model, another generator populates the *client data model* of the XVCML model with appropriate process data information.

The benefit of this integration is obvious: (1) Most engineering data can be reused which a) decreases the overall engineering effort. Furthermore, it b) ensures consistency between the HMI and the information given in the documentation that was prepared throughout the planning stage, as required by VDI/VDE (1997 – 2005). (2) Operators — the end users of the HMI — can already participate in early design stages, which has two significant advantages: a) First, based on their feedback, the user interface can be better tailored to their needs, which results in a more reliable plant operation. b) Second, the operator training can already occur during the development of the plant. For this purpose, mockups representing the final look and feel can be derived from the XVCML model, even though it has not yet been populated with concrete process data information. This reduces the overall time between the planning of the plant and its commissioning.

7.2 Useware Engineering Process

The previous section concludes with early participation of operators in engineering procedures as they are primarily beneficial to ensure reliable plant operation. The *Useware Engineering* [Zühlke, 2004] process explicitly proposes to accompany each phase during the development of user interfaces for technical systems by an end-user evaluation phase. Each development phase describes the user interface at a different level of abstraction by making use of a dedicated modeling language. Figure 7.2 demonstrates this process: Based on a task model that was deduced in the *analysis phase*, a *UseML* [Meixner, 2010; Meixner, Seissler, and Breiner, 2011] model is created in the *structuring phase*. It represents a hierarchically ordered structure of tasks. This forms the basis for the dialog model that is created in the *design phase*. In this phase, abstract interaction elements and their interrelationships are formalized using the description language *DISL* [Schäfer, 2007]. This abstract user interface model is concretized into a UIML [Abrams et al., 1999] model which is a concrete user interface model, but resides in the design phase. The *realization phase* creates the user interface to be deployed to particular runtime artifacts. As each phase is accompanied with a user *evaluation phase*, the respective feedback

can be taken into consideration as early as possible, leading to user interfaces with a high usability.

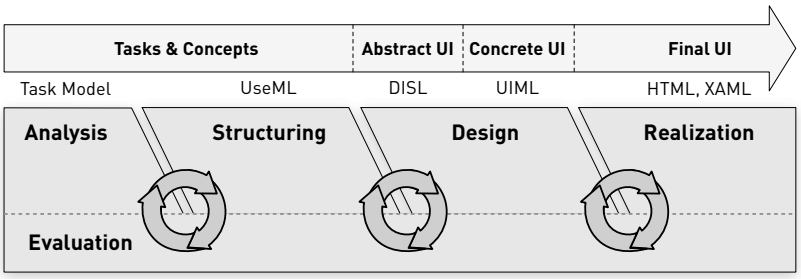


Figure 7.2: Phases of the Useware Engineering process classified by means of the CAMELEON REFERENCE FRAMEWORK. Each development phase comes with a particular modeling language at a different level of abstraction.

According to Meixner (2010); Meixner, Seissler, and Breiner (2011), the Useware process can be mapped to the CAMELEON REFERENCE FRAMEWORK, as depicted in Figure 7.2. Consequently, in [Hager et al., 2011], the MOVISA language was integrated in the design phase of the Useware process. Based on a DISL model, transformations generate the MOVISA model which is, in turn, the basis for transformations to final runtime artifacts. However, when transforming an abstract model to a concrete one, an element of the source model has possibly more than one counterparts in the target model. To solve this *mapping problem* [Clerckx, Luyten, and Coninx, 2004], interactive transformations were introduced: If the transformation encounters elements with more than one possible transformation targets, it will ask the developer for a solution.

The Useware process is an iterative process: When identifying issues while evaluating the models of a particular development phase, the developer goes back to the preceding phase to fix this issue. After this, the tools which establish the transition to the next phase are reapplied. For the particular case, this means that the interactive transformations might possibly solve the same mapping problems as encountered in the iteration step before. Consequently, a *Persistent Transformation Mapping* (Petmap) infrastructure was introduced. It stores all decisions being made during a transformation to replay them in subsequent iteration steps. This procedure is illustrated in Figure 7.3: The first transformation (①) needs developers to solve ambiguous mappings (②) which will be stored in the Petmap. After modifying the source model (③), a subsequent transformation (④) uses the information of the Petmap (⑤) and, hence, the interactions are only reduced to new DISL elements (⑥).

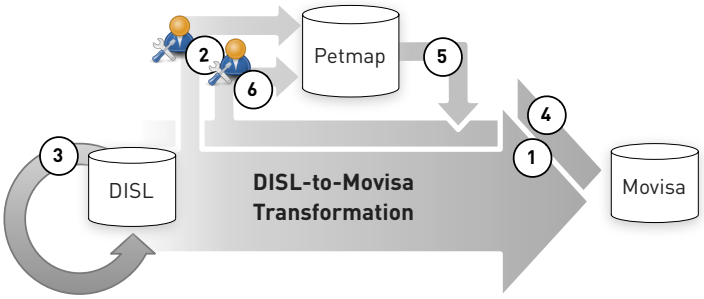


Figure 7.3: Petmap: Reducing developer interaction when applying interactive transformations in iterative development processes (from [Hager et al., 2011]).

The benefit of integrating MOVISA in the Useware engineering process is a systematic user interface design that is based on models of different levels of abstraction. More importantly, however, is that operators participate in each design stage by evaluating the respective artifacts and by giving feedback on the improvement of the particular models. The challenge was to bridge the abstraction gap through interactive transformations that store the decisions into another model, the Petmap. Consequently, developers directly participate in the transformation process and, thus, elements can automatically be created in conformance with expectations, making a manual post-treatment obsolete.

7.3 Flepr: Flexible Transformation Based Workflows

Concluding the previous section, Section 7.2, the integration of MOVISA into the Useware engineering process pointed to one drawback: Fixing issues, identified when evaluating the respective models, has to always take place in the preceding development phase to preserve consistency between models. In the strictest sense, however, incorporating the evaluation feedback while preserving consistency between all models must always start at the most abstract level and then be rolled out appropriately to the particular more concrete model. To overcome the drawback of tedious and repetitive modeling endeavors, the *FLEPR* approach was proposed in [Hennig et al., 2011]. FLEPR enables addressing issues, identified during user evaluations, at the level where they actually were identified. Once the evaluation feedback was completely incorporated on one level, these modifications were synchronized with the other models to keep them in a consistent state.

The motivation for this work was the integration of the MOVISA language into the entire MBUID process, as introduced in Section 2.2, as well as the combination

of it with the *User Centered Design* (UCD) process. Very similar to the Useware process, the UCD is an iterative process: Ideas and concepts of the UI will be expressed in abstract UI models and concretized into prototypes that are tangible for end-users in an early point in time. According to the feedback of the end-users after evaluation, the prototype will be modified and thereafter reevaluated (see Figure 7.4). In MBUID, the evaluation and thus, the modification can take place at the concrete level where the user interface is getting more tangible. However, altering user interface models at the concrete level might not influence the abstract models, which would lead to model inconsistencies.

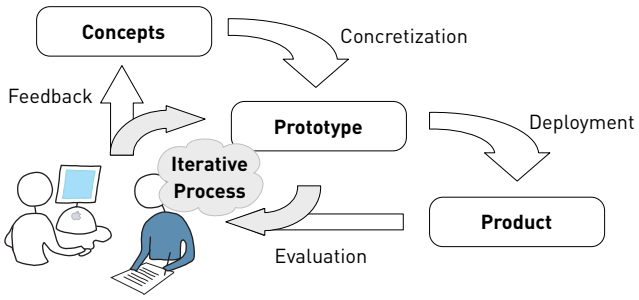


Figure 7.4: Basic principle of the *User Centered Design* process.

Two main challenges were identified when combining the MBUID procedure with the UCD: (1) Solving ambiguities when transforming abstract models to more concrete ones, as already discussed in Section 7.2 (mapping problem), and (2) preserving consistency across models of different levels of abstraction. These challenges were tackled in this work using the following project setup: Modeling languages of different but adjacent levels of abstraction were used, whereas MOVISA formed the concrete (CUI) part and CAP3 [Van den Bergh, Luyten, and Coninx, 2011] the more abstract user interface (AUI) modeling language. CAP3 is a domain specific modeling language for abstract user interface models that provides both an abstract syntax notation based on *Eclipse EMF* and a graphical concrete syntax notation based on *GMF*. Both languages were chosen for this project for two reasons: (1) Both provide an EMF-based abstract syntax fostering convenience when creating the model transformations. (2) Both feature a concrete syntax notation which enables high-fidelity prototyping.

Both modeling languages and, thus, both levels of abstraction are additionally characterized by the users being involved in the respective design process. *Interaction Designers* and *Information Architects* are familiar with usability standards (e.g. [DIN, 1999]), dialog design (e.g. [DIN, 1996] or particularly in the automation domain: [VDI/VDE, 1997 – 2005]), and/or information representation (e.g. [DIN,

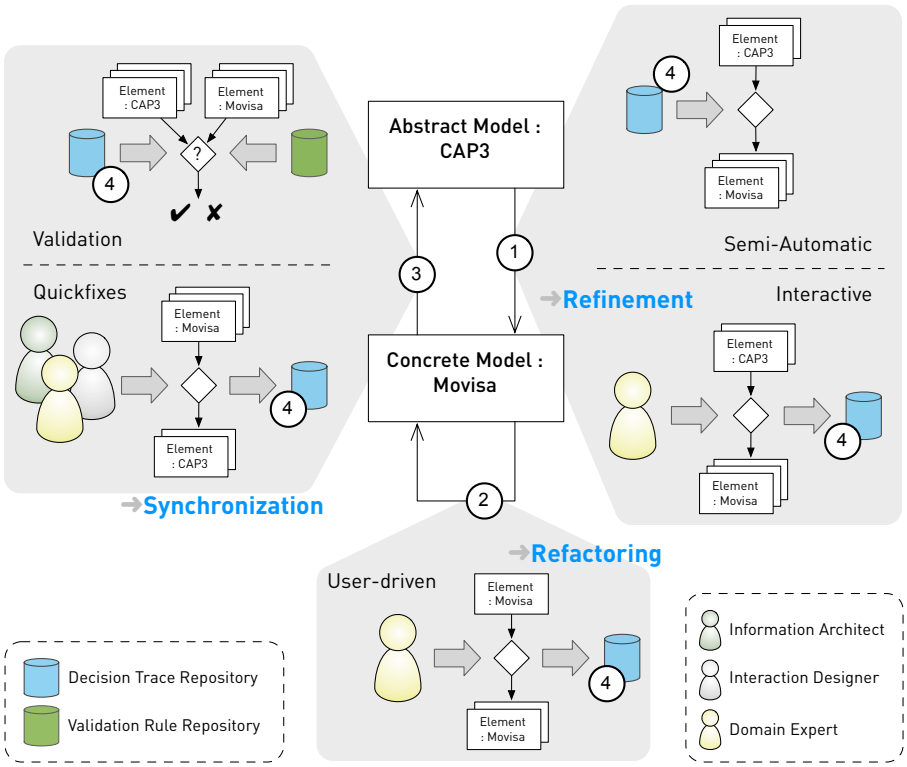


Figure 7.5: FLEPR: The overall concept showing details for each develop step, namely ① Model Refinement, ② Model Refactoring, and ③ Model Synchronization. This figure also shows which kind of users are involved in the particular development phase.

1998]). Thus, these users are responsible for creating and maintaining the abstract UI model CAP3. *Domain Experts* are the application engineers with special knowledge about the technical process and the physical backgrounds, possible events and hazardous situations as well as about the product to be produced. As MOVISA was designed to capture specific knowledge of the automation domain, this user role is responsible for creating and maintaining the CUI model with MOVISA. *End users* are the operators indented to monitor and operate the technical process through the visualization solution being under development. Hence, this user role is in charge of evaluating the user interface prototypes.

To establish the UCD process, the FLEPR concept is based upon three different kinds of model transformations and techniques to track decisions and modifications, as depicted in Figure 7.5 and explained in the following:

① *Model Refinement* transformations take the CAP3 model as a source model

and produce a MOVISA model as a transformation target. As this kind of transformation has to bridge the abstraction gap, as discussed in Section 7.2, they are interactive and, thus, require the *domain expert* to guide the transformation process.

② *Model Refactoring* transformation are used to improve or to modify one model: Once the MOVISA model is produced, the *end user* comes into play by evaluating the actual design and by giving his or her feedback to the domain expert. While changing an element's position or size is a rather trivial task, it can be very tedious to replace an element by another one, since all the representation, animation, and interaction properties must be preserved and then adopted. *Refactoring Transformations* can support this task. Kolovos et al. (2007) define them as *update transformations in the small:* They "are applied in a model-driven manner on model elements that have been explicitly selected by the user."

③ *Model Synchronization* transformations manage the consistency between two or more models if one of these models was altered: After the MOVISA model was modified due to *end user* feedback, the CAP3 model has to reflect these modifications. For this purpose, *intra- and inter-model validation* were introduced to ensure the semantical correctness between the MOVISA and the CAP3 model. If inconsistencies have been identified, solving them would incorporate decisions whether to roll back the modifications or to update the counterpart model. In both cases the user will be supported by *user-driven update transformations*.

④ *Traceability.* The connection between source and target elements as well as each intervention of the respective users in the different transformation processes are traced in a separate *decision trace repository*, the *FleprMap*. This model forms the basis to reproduce refinement transformations, to roll back refactoring transformations, and to ensure the proper functioning of the synchronization transformations. Hence, the *FleprMap* is the core of the FLEPR concept.

The benefit of this work — integrating MOVISA into the MBUID process and applying a UCD process — is that the employment of MOVISA can accompany with recommended development processes: (1) MOVISA can be part of the MBUID process which uses models of different levels of abstraction to provide a systematic user interface engineering. (2) Being part of the UCD process and, thus, enabling operators to give their feedback to the user interface designs already in early development stages leads to more robust user interfaces and, thus, to more a reliable operation of the technical process. Another meaningful outcome of this work is, however, the

investigation of the different kinds of transformations. Not only do they enable transforming a source model into a target model, more important are the model update and modification tasks, which are supported by transformations to reduce repetitive decisions. Furthermore, the model synchronization transformations allow to keep several models in a consistent state if one of them was altered.

7.4 Conclusions

Olsen Jr. (2007) claims that with *Reduce Development Viscosity* a good user interface tool reduces the time it takes to create a new solution. This chapter showed that MOVISA can be part of established user interface design procedures. Furthermore, several parts of the model can automatically be populated from the models of previous engineering phases through the low-fidelity interface of the MOVISA modeling workbench, reducing the overall effort to create a new solution and fostering consistency across development stages. Particularly, Section 7.1 showed that the models of the MOVISA language can be created from plant engineering data, as required by Requirement 5 on page 27. Section 7.2 and Section 7.3 demonstrated that integrating MOVISA into iterative processes, as required by Requirement 6 on page 28, and already involving the end users in early design stages can lead to more robust user interfaces that are characterized by reliable operations of the technical process, also in extreme situations.

Moreover, when using the MOVISA tool together with CAP3 in a combined workflow, different kinds of experts are enabled to participate in the design process, each with their known terminology. This accompanies with the *lower skill barrier* claim [Olsen Jr. 2007], which means that with a good toolkit design, more people with different skills can effectively create new applications.

The previous sections also showed that many kinds of transformations can be applied to a MOVISA model. They are (1) refinement transformations for creating a MOVISA model, (2) refactoring transformations for modifying a MOVISA model, and (3) synchronization transformations for preserving consistency of a MOVISA model with another model after altering one of them. These transformations play an important role in supporting modelers in repetitive, tedious, and/or error-prone tasks, as they provide manifold possibilities to process models. The concept behind these transformations is of general nature and can also be applied in other model based engineering processes. Finally, the effort to create and to maintain these transformations must not be underestimated.

Chapter 8

Conclusions

To summarize the previous chapters, successfully exploiting a *Domain Specific Language* is an exhausting endeavor due to (1) determining and analyzing the *Target Domain*, (2) formalizing the domain abstractions into a *Language Model* with having possible extensions in mind, (3) designing an efficiently *Concrete Syntax* notation, (4) providing an easy-to-use *Modeling Workbench* to use the language and to maintain models, and, finally, (5) integrating this modeling workbench into tool chains currently being used in industry. A study to clarify when this effort will be amortized has not been carried out, because not only did the efforts require to exploit a DSL should be an estimate of its success, but also more importantly is that a DSL fosters reuse and, thus, sustainability of system designs through separating operative characteristics from their platform specific realizations. By dealing with a problem at a higher level of abstraction, model verification, model validation, model-integrity checks, and even model testing can be addressed in this pure functional solution space. Together with tested and possibly certified transformations, this might result in better platform specific solutions — the source code. Section 8.1 presents how these aspects were addressed by meeting the *Requirements* stated in Chapter 2. Section 8.2 discusses recommendations for future work.

8.1 Achievements and Contributions

Section 2.5 states six major requirements that directed the work discussed throughout the previous chapters. In detail, this thesis contributes a domain specific modeling workbench for a sustainable development of visualization solutions to monitor and operate the operative states of technical processes. This section summarizes these achievements and contributions by explicitly responding to these requirements:

Requirement 1: CUI Modeling Language. In Chapter 3, the *Domain Specific Language* MOVISA was worked out by tailoring its language model to the domain "HMIs to monitor and operate technical processes with respect to the field of production automation", as stressed by Requirement 1 on page 22. Particularly, it provides an appropriate core language model defining modeling elements for expressing (1) user interface components with appropriate *representation*, *animation*, and *interaction* properties, (2) solid abstractions to industrial automation specific process communication means, (3) application specific functionality through an *Executable UML* realization. Chapter 6 proves MOVISA to feature sufficient expressiveness through successfully performing case studies of its application domain. Hence, with this modeling language, the *operative characteristics* can be captured completely separate from its technical realization. This ensures sustainable visualization solution designs. To better manage the complexity of the comprehensive *Core Language Model* and to foster its maintenance, the following measures were introduced:

(1) *Forming Subdomains:* As the modeling language needs to provide different views on a visualization solution — user interface view, process data view, and custom functionality view — and each view addresses another concept, they were encapsulated in separate submodels. Hence, each of these different concepts can be specified with the most efficient means for the given domain abstraction.

(2) *Drawing relationships between submodels through signals:* With introducing a simple, but powerful interconnection concept to establish relationships between elements of different submodels, the permutations of relationships were reduced to a minimum. Furthermore, when adding new language elements to maintain the language (comp. Section 6.1.2, where a new server element was necessary), less interdependencies have to be taken into account by implementing against a simple interface.

(3) *Extension points:* To remain complexity at a manageable level, the core language model focuses on the relevant domain abstractions. To capture borderline aspects, addressing very specific requirements, or domain aspects that are relevant as of a future point in time, reasonable *extension points* have been provided: a) While the ALGORITHM MODEL itself allows to address specific requirements, the BOUNDARY goes beyond what can be captured by models. b) The PRESENTATION MODEL allows to preserve and reuse complex user interface widgets. c) The CLIENT DATA MODEL provides extensions through the GENERIC SERVER element.

Requirement 2: High-Fidelity Concrete Syntax Notation. Chapter 4 discusses a high-fidelity notation for MOVISA models which combines textual and graphical artifacts, according to the aspect of the visualization solution to be expressed. Textual notations are more efficient when it comes to capturing data structures or plain configuration data. Graphically concrete syntax notations are better suited to depict data and control flows or when having whiteboard discussions. With this, a high-fidelity modeling interface can be provided enabling domain experts to exploit MOVISA models close to the problem space. Consequently, Requirement 2 on page 24 is met with respect to the discussion in "Expressive Match" of Section 6.2.1. However, the more powerful and comprehensive the language is, the higher are the initial hurdles and the more exhaustive the development can become for successfully exploiting the language. It is up to an efficient tool to hide most of the complexity of the language. Through capturing different viewpoints on a problem by respective submodels, modelers are always only faced with relevant information: With reasonably partitioning the model, as explained in Section 5.6, irrelevant parts of it are hidden. Furthermore, the MOVISA modeling workbench, as discussed in Chapter 5, provides wizards to assist modelers during repetitive and tedious tasks. In this respect, Section 7.3 presents *horizontal refactoring transformations* which allow to easily replace a user interface element by another one while adopting its existing configuration. Together with providing convenient menu controls to create models and invoke transformations, these aspects also address what Requirement 2 on page 24 has asked. Hence, providing an high-fidelity concrete syntax notation and an efficient modeling tool is a cornerstone to raise the efficiency and reduce the development costs.

Requirement 3: Model Verification Tool. Raising the development efficiency also means to reduce the time to detect failures in models. For this purpose, Section 3.3 explores on verification rules to express constraints on the language that cannot be expressed by the core language model. Model-integrity checks of any complexity help the modeler to identify faults in early design stages. All constraints can be as complex as necessary, since they can be formulated using the Java programming language, as discussed in Section 5.2. Even though this verification framework forms a sound basis to meet Requirement 3 on page 25 and increase the reliability of the resulting solution, the efficiency of these error detection means strongly depends on the extent of the constraints formulated.

Requirement 4: Vertical and Horizontal Transformations. The more automatic means can be provided, the less effort is needed to produce a visualization

solution and the higher is the efficiency. The CAMELEON REFERENCE FRAMEWORK proposes to apply vertical transformations to establish a transition between models and platform-specific source codes at FUI level. An important outcome is that with a clear boundary around the applications to be built and precise language semantics (comp. Section 3.4) leaving no room for interpretations, these transformations are capable of completely automatically producing all required platform specific runtime artifacts. In Chapter 6, they were successfully applied to the models of the case studies. It is a different situation, however, if horizontal transformations establish a MOVISA model in another context of use, as proposed by the CAMELEON REFERENCE FRAMEWORK. These transformations cannot be completely automatically realized, as the user interface components possibly need another layout. In a semi-automatic manner, the modeler annotates the model and, thus, instructs the transformation how to treat individual components. "CUI-to-CUI Transformation" of Section 5.3.2 presents a framework for this purpose. Even though a manual post-treatment of the target model is necessary, the transformation support brings a considerable reduction of the amount of work involved.

Even though transformations are required to be developed for each target technology, many aspects can be reused. For instance, the graph analysis algorithm, as discussed in Section 5.3.2, could completely be reused in every target technology due to separating *fixed operative characteristics* from platform related aspects. Consequently, Requirement 4 on page 26 is met, which leads to the following benefit: When separating operative characteristics from their technical realization, as achieved by the MOVISA modeling language and modeling workbench, both aspects can separately be developed. Based on a precise language semantics description, a common basis — an interface contract — between both language aspects exists. Consequently, different experts, whether they are domain experts or platform specialists, can concentrate on their assigned tasks simultaneously. This, in turn, does not only reduce the overall project time, it also fosters the work in multidisciplinary teams.

Requirement 5: Low-Fidelity Tool Interface. Requirement 5 on page 27 asks to reuse existing data from previous engineering stages. By including the predecessor language of MOVISA into the *autoHMI* development procedure (comp. Section 7.1), data from engineering tools populate the concrete models: Hierarchy and structural information form the basis for an initial PRESENTATION MODEL configuration. Information about actual plant equipment is used to create the TECHNICAL DATA PERSPECTIVE. Also when using the MOVISA language in the *Useware Engineering*

process (see Section 7.2) or together in other user-centered engineering procedures (comp. Section 7.3), the data stored in the previous models can be populated using the low-fidelity interface of the MOVISA modeling workbench and prepared to serve as input for the MOVISA models. With these works, even the claim of Martinie and Palanque (2011) — including a new concept into established workflows — could be met.

Requirement 6: Iterative Development Procedures. Model based user interface development procedures exploit models at different levels of abstraction, whereas the MOVISA language is classified in more concrete levels, such as the CUI level of the CAMELEON REFERENCE FRAMEWORK. In conjunction with incremental and iterative design procedures, which mainly concern improving user interface prototypes according to the feedback of evaluation phases, a notable challenge can be identified: The models can become inconsistently across the abstraction levels. To overcome this challenge, Section 7.3 proposes *Synchronization Transformations* which support the modelers with keeping their models in a consistent state. (This synchronization does not take the runtime artifacts into account. Synchronizing modifications, performed at this level, back into the model level remain a challenge in software engineering.) However, many modifications at a more concrete level do not have any influence on the respective abstract models. Hence, these modifications need to be preserved when applying *Refinement Transformations* in another iteration. For this purpose, Section 7.2 proposes the *PetMap* and Section 7.3 the *FleprMap*. Both are specific decision trace models to capture all modifications to replay or to revert them. With this framework, Requirement 6 on page 28 is met and, therewith, the technical foundation to apply user-centered design procedures has been laid.

Overall Result. Myers, Hudson, and Pausch (2000) argue that automatic and model-based techniques suffer from the unpredictable production of runtime artifacts and from being too difficult for developers, as they have to learn a new language. While they use this language, they are not able to foresee the final result. Due to these still unsolved challenges, Meskens, Luyten, and Coninx (2010) propose another approach: *D-Macs* enables developers to *record* their design actions while using their preferred toolkits and *replay* these actions in another toolkit. However, with this tool, developers must always learn the programming language of the target toolkit.

In this thesis, the model-based technique MOVISA was worked out which is based on *Domain Specific Languages*. MOVISA overcomes the aforementioned

challenges by featuring an efficient and high-fidelity concrete syntax notation being tangible enough to enable developers to see the final result already at the time of modeling. In combination with a precise language behavior definition, automatic transformations produce the respective runtime artifacts. Even though a MOVISA model completely captures the operative characteristics of a visualization solution, real platform independence depends on the capabilities of the intended target platform. A platform cannot be used if it does not provide a required infrastructure. With an adequate deployment configuration, the range of *capable* platforms can be enlarged. This knowledge is captured by the transformation, which are therefore a key factor for the successful employment of a model driven procedure. The MOVISA tool provides different kinds of transformations — for *refining*, *refactoring*, or *synchronizing* models — which can be combined very flexibly. However, developing and maintaining efficient and flexible transformations are very demanding processes: Besides capturing all platform related concerns, they contain all the *fixed operative characteristics* of a system and, therewith, all expert knowledge. Thus, special attention should be payed to them. As this goes beyond the scope of this thesis, recommendations for future research will be discussed in Section 8.2.

To sum up, this thesis contributes the domain specific modeling workbench MOVISA containing in particular:

(1) a domain specific language enabling sustainable visualization solution designs at CUI level;
(2) a high-fidelity concrete syntax notation fostering domain experts to work and to think in their domain;
(3) an extensible model verification tool ensuring correctness of models already in early design stages;
(4) a transformation tool containing both vertical and horizontal transformations;
(5) a low-fidelity tool interface fostering continuous tool chains; and
(6) a transformation based framework enabling iterative design procedures.

With this, textually and graphically concrete syntax notations were combined, constraints were expressed using different languages, and both declarative and imperative transformation languages were used simultaneously. Hence, it can be argued that the success of exploiting model driven engineering procedures depend on the degree of freedom when combining different tools and technologies.

8.2 Future Work

Although this thesis answers important research questions and meets the requirements of Section 2.5, several open aspects that should be tackled in future research could have been identified. These are discussed in detail in the following.

Transformations

Concluding the previous section, transformations form a cornerstone for the success of the model driven development procedures. Hence, a reasonable modularization was introduced which in turn could be proven to be advantageous when it comes to maintaining the transformations. However, as many different deployment configurations need to be taken into account, as discussed in Section 5.3.3 and in Chapter 6, a flexible approach to compose transformation rule modules should be striven for. Many authors presented tools to compose transformations: Vanhooff et al. (2007) proposed the transformation composition framework *UniTI*, Heidenreich, Kopcsek, and Aßmann (2010) presented *TraCo*, and Kleppe (2006) introduced *MCC*. All approaches share that they use an *external* composition, where models are handed over from one transformation to another one. This is particularly useful for runtime configurations of visualization solutions as shown in Figure 6.5: It consists of different independent components (*Alarm Management*, *Historical Data*) and the model was transformed to visualization clients of several target technologies. To flexibly compose a single transformation (those producing individual component, e.g. the *Standalone Visualization Solution* in Figure 6.5) of reusable modules, *internal* composition is required. Wagelaar et al. (2011) presented a first concept for an internal composition mechanism. It should be explored in future research whether these composition frameworks and concepts can be employed to bring more flexibility into the transformation process.

Sharing these reusable transformation modules between modelers leads to an even more powerful tool. Hence, a centralized repository might be established with a community of modelers build around.

Modeling Security Aspects

IT security also is an important topic, if not particularly, in industrial automation. Bettenhausen and Morr (2007) stated that IT security in automation is not so much a technological challenge as an organizational one. Hence, both specifications [NAMUR, 2006] and [VDI/VDE, 2007] are devoted to organizational security frameworks within the scope of industrial automation. Obviously, technological

security needs to be considered in visualization solutions and, thus, future research must take into account how to integrate these concerns into a MOVISA model.

An idea is to provide a security model describing, among others, access rights on data items and capturing required cryptographic keys for authorization and encryption of the message channel. However, applying security concerns, in particular those limiting or granting operators access to process values, possibly causes side-effects in one visualization solution. For instance, when limiting write access to particular DATA ITEMS, the faceplates to modify these DATA ITEMS must reveal this limitation to the operator. If reading the DATA ITEM is not allowed, then the respective user interface components must disappear to prevent the operator from thinking that something might went wrong. However, this might also have an affect on other user interface components.

Due to these possibly complex interdependencies, modelers need to be able to check whether the resulting visualization solution is working as expected when formulating a security policy. The idea is to encapsulate these security aspects into a separate model. If required, these security information is woven into the MOVISA model in an *aspect-oriented way*, e.g. as proposed by Hovsepyan et al. (2007). This process is very similar to the Model-to-Model transformation when translating a MOVISA model into another context-of-use. In this case, however, user interface elements are temporarily removed depending on the actual access rights so that the modeler can get an impression already at an abstract model level of how the user interface looks like for individual operators. Furthermore, with the security aspects woven into the MOVISA model, resulting interdependencies can also be subject to a model verification phase.

Automatically (Re-)Layouting User Interfaces Components for Small Screens

Section 5.3.2 discusses a *Model-to-Model* transformation to transfer a particular MOVISA model into another context of use, on CUI level. Even though development efforts can be reduced by means of this transformation because it performs many required modifications automatically, a manual post-treatment is necessary. Future research should be devoted to realize more efficient layout algorithms to increase the level of automation. Roscher et al. (2011) propose to use a layout model dedicated to capture modeler's preferences. A layout engine reads this additional model at runtime. Based on the actual device, it applies these particular constraints to the user interface elements. Raneburger et al. (2011a) use different optimization strategies (maximum usage of the available space, minimum navigation clicks, and

minimum scrolling) to automatically generate user interfaces optimized for small screens.

This current research shows that automatic layouting is a very challenging endeavor. However, the more information an algorithm has available, the more likely is its success and, thus, the better can the results be reinforced. In particular, the following information is relevant for an automatic (re-)layouting and should be taken into account:

- VDI/VDE (1997 – 2005) contains many informal information about how a user interface in industrial automation should look like. Hence, it should be formalized and used by the transformation.
- Modeler's own preferences should be taken into account, as it has already happened in [Roscher et al., 2011] through their layout model.
- Formalized user interface patterns should feed the transformation, as proposed by Seißler, Breiner, and Meixner (2011).
- Company-wide style guides can be formalized and used within the transformation.

Model Instance Library

To further on foster reuse, parts of models can be shared among different projects, different modelers, or companies. MOVISA enables to reuse user interface components through the COMPLEX UI COMPONENT (comp. Figure B.21) and its concrete elements (comp. Figure B.41). However, in the current development state, these components must be manually extracted from the model. To encapsulate even this complexity, future work should be devoted to mechanisms which allow the MOVISA modeling workbench to extract, import, and share these complex elements. Using a centralized repository, industry-specific complex components can be made available within a community, thus increasing the efficiency of the entire MOVISA tool. Furthermore, when additionally specifying the semantical meaning of individual complex components, a vision could be to automatically choose components from this repository for a given problem.

References

Abrams, Marc, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster (1999). "UIML: an appliance-independent XML user interface language". In: *Computer Networks* 31.11-16, 1695–1708 (cit. on pp. 11, 133).

Abstract Solutions (2011). *PRODUCTS — Modelling :: iUML Modeller and Simulator*. Website, last visited: November 22nd, 2011. URL: http://kc.com/PRODUCTS/iuml/index.php (cit. on p. 81).

Aquino, Nathalie, Jean Vanderdonckt, Nelly Condori-Fernández, Óscar Dieste, and Óscar Pastor (2010). "Usability Evaluation of Multi-Device/Platform User Interfaces Generated by Model-Driven Engineering". In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '10, 30:1 – 30:10 (cit. on p. 4).

Bailey, David and Edwin Wright (2003). *Practical SCADA for Industry*. IDC Technology. Elsevier. ISBN: 9780750658058. URL: http://books.google.de/books?id=rylG0sJYREoC (cit. on p. 2).

Barnett, Jim, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T.V. Raman, Klaus Reifenrath, and No'am Rosenthal (Apr. 2011). *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. W3C Recommendation. URL: http://www.w3.org/TR/2011/WD-scxml-20110426/. W3C (cit. on p. 122).

Benyon, David, Thomas Green, and Diana Bental (1999). *Conceptual Modeling for User Interface Development*. Practitioner Series. Springer (cit. on pp. 11, 14).

Bettenhausen, Kurt D. and Wolfgang Morr (2007). "Informationssicherheit in der Automatisierung — Mehr als eine technologische Herausforderung". In: *atp* Heft 4, pp. 76 – 79 (cit. on p. 146).

Blankenhorn, Kai and Mario Jeckle (2004). "A UML Profile for GUI Layout". In: *Object-Oriented and Internet-Based Technologies*. Ed. by Mathias Weske and Peter Liggesmeyer. Vol. 3263. Lecture Notes in Computer Science. Springer, pp. 315 – 333 (cit. on p. 22).

Botterweck, Götz (2007). "A Model-Driven Approach to the Engineering of Multiple User Interfaces". In: *Models in Software Engineering*. Volume 4364/2007. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, pp. 106–115 (cit. on pp. 22, 51).

Braune, Annerose, Stefan Hennig, and Torsten Schaft (2007). "XML-based monitoring and operating for Web Services in automation". In: *Proceedings of the 5th IEEE International Conference on Industrial Informatics, INDIN, 2007*. Vol. 2, pp. 797 – 802 (cit. on p. 132).

Bry, Francois, Bernhard Lorenz, Hans Jurgen Ohlbach, Martin Roeder, and Marc Weinberger (2008). "The Facility Control Markup Language FCML". In: *Proceedings of the Second International Conference on Digital Society*. ICDS '08. IEEE Computer Society, 117 — 122 (cit. on p. 32).

CFR 21 (1997). *Title 21 — Food and Drug Administration. Part 11 Electronic Records; Electronic Signatures*. Code of Federal Regulations Title 21. 62 FR 13464 (cit. on p. 50).

Calleros, Juan M. González, Gerrit Meixner, Fabio Paternò, Jaroslav Pullmann, Dave Raggett, Daniel Schwabe, and Jean Vanderdonckt (May 2010). *Model-Based UI XG Final Report*. W3C Incubator Group Report. URL: http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504 (cit. on pp. 11, 12).

Calvary, Gaëlle, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt (2003). "A unifying reference framework for multi-target user interfaces". In: vol. 15, pp. 289–308 (cit. on pp. 12, 13, 98).

Cassandras, Christos G. and Stéphane Lafortune (1999). *Introduction to Discrete Event Systems*. Kluwer Academic Publishers (cit. on p. 1).

Chen, Kai, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson (2005). "Semantic anchoring with model transformations". In: *Model Driven Architecture: Foundations and Applications (ECMDA-FA 2005)*. Vol. 3748. Lecture Notes in Computer Science. Springer, 115 — 129 (cit. on p. 17).

Clerckx, Tim, Kris Luyten, and Karin Coninx (2004). "The Mapping Problem Back and Forth: Customizing Dynamic Models while preserving Consistency". In: *Proceedings of the 3rd annual conference on Task models and diagrams*. TAMODIA '04. ACM, 33 — 42 (cit. on p. 134).

Cormen, Thomas H., Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson (2001). *Introduction to Algorithms*. 2nd edition. McGraw-Hill Higher Education (cit. on p. 96).

Coutaz, Joëlle (2010). "User interface plasticity: model driven engineering to the limit!" In: *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '10. Berlin, Germany: ACM, pp. 1–8 (cit. on p. 12).

Czarnecki, Krzysztof and Simon Helsen (2003). "Classification of Model Transformation Approaches". In: *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture* (cit. on p. 94).

DIN (1993). *Control technology; graphical symbols and identifying letters for process control engineering; symbolic representation for functions (DIN 19227-1:1993-10)*. DIN 19227-1 (cit. on p. 84).

– (1996). *Ergonomic requirements for office work with visual display terminals (VDTs) — Part 10: Dialog principles (DIN EN ISO 9241-10)*. DIN EN ISO 9241-10 (cit. on p. 136).

– (1998). *Ergonomic requirements for office work with visual display terminals (VDTs) — Part 8: Requirements for displayed colours (DIN EN ISO 9241-8)*. DIN EN ISO 9241-8 (cit. on p. 136).

– (1999). *Ergonomic requirements for office work with visual display terminals (VDTs) — Part 11: Guidance on usability (DIN EN ISO 9241-11)*. DIN EN ISO 9241-11 (cit. on p. 136).

– (2006). *Function blocks — Part 1: Architecture (IEC 61499-1:2005)*. DIN EN 61499-1 (cit. on p. 42).

Daneels, A. and W. Salter (1999). "What is SCADA?" In: *Interantional Conference on Accelerator and Large Experimental Physics Control Systems* (cit. on pp. 2, 9).

Date, C. J. and H. Darwen (2007). *Databases, types and the relational model: the third manifesto*. Addison-Wesley (cit. on pp. 41, 46).

Demirezen, Zekai (2009). "Semantic Framework for DSLs". In: *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. OOPSLA '09. ACM, 833—834 (cit. on p. 17).

Digitronic (2011). *Webvisualisierung*. Website, last visited: September 23rd, 2011. URL: http://www.digitronic.com/digiweb/home.html (cit. on p. 10).

EEMUA (2007). *Alarm Systems — A Guide to Design, Management and Procurement*. EEMUA Publication No 191, Edition 2. The Engineering Equipment and Materials Users' Association (cit. on pp. 49, 72).

EMFText (2011). *EMFText Concrete Syntax Mapper*. Website, last visited: September 30th, 2011. URL: http://www.eclipse.org/modeling/emf/?project=validation (cit. on pp. 20, 104).

Eberle, Stephan (2007). "Adaptive Internet Integration of Field Bus Systems". In: *IEEE Transactions on Industrial Informatics* 3.1, pp. 12–20 (cit. on p. 43).

Eclipse (2011a). *EMF Validation Framework*. Website, last visited: September 28th, 2011. URL: http://www.eclipse.org/modeling/emf/?project=validation (cit. on p. 19).

– (2011c). *Eclipse Modeling Framework (EMF)*. Website, last visited: September 28th, 2011. URL: http://www.eclipse.org/modeling/emf (cit. on p. 16).

– (2011b). *Eclipse*. Website, last visited: November 17th, 2011. URL: http://www.eclipse.org (cit. on p. 91).

– (2011d). *Epsilon Generation Language*. Website, last visited: November 18th, 2011. URL: http://www.eclipse.org/gmt/epsilon/doc/egl/ (cit. on p. 102).

– (2011e). *Epsilon Object Language*. Website, last visited: November 18th, 2011. URL: http://www.eclipse.org/gmt/epsilon/doc/eol/ (cit. on p. 102).

– (2011f). *Epsilon Transformation Language*. Website, last visited: November 18th, 2011. URL: http://www.eclipse.org/gmt/epsilon/doc/etl/ (cit. on p. 102).

Eclipse (2011g). *Epsilon Validation Language*. Website, last visited: September 28th, 2011. URL: http://www.eclipse.org/gmt/epsilon/doc/evl/ (cit. on pp. 19, 93).

– (2011h). *Graphical Modeling Project (GMP)*. Website, last visited: September 30th, 2011. URL: http://www.eclipse.org/modeling/gmp/ (cit. on pp. 20, 103).

– (2011i). *Xtext Documentation*. Website, last visited: November 20th, 2011. URL: http://www.eclipse.org/Xtext/ (cit. on p. 103).

Eisenstein, Jacob, Jean Vanderdonckt, and Angel Puerta (2001). "Applying Model-Based Techniques to the Development of UIs for Mobile Computers". In: ACM Press, pp. 69–76 (cit. on p. 10).

Epsilon (2011). *Eclipse Epsilon*. Website, last visited: September 27th, 2011. URL: http://www.eclipse.org/gmt/epsilon/ (cit. on pp. 16, 102).

Feldt, Kenneth C. (2007). *Programming Firefox: Building Applications in the Browser*. 1. ed. O'Reilly (cit. on p. 14).

Fowler, Martin (2011). *Domain-Specific Languages*. Addison-Wesley (cit. on pp. 17, 21, 42, 81, 88, 94, 103, 130).

Frankel, David S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley (cit. on pp. 18, 19).

Frey, Georg (2008). "Distributed Control Applications with IEC 61499". In: *atp* Heft 12, pp. 56 – 61 (cit. on p. 42).

GME (2011). *GME: Generic Modeling Environment*. Website, last visited: September 30rd, 2011. URL: http://www.isis.vanderbilt.edu/Projects/gme/ (cit. on p. 20).

Goodger, Ben, Ian Hickson, David Hyatt, and Chris Waterson (2011). *XML User Interface Language (XUL) 1.0*. Ed. by David Hyatt. Website, last visited: September 27th, 2011. URL: http://www-archive.mozilla.org/projects/xul (cit. on p. 14).

Green, T. R. G. (1989). "Cognitive dimensions of notations". In: *People and Computers V*. Ed. by A. Sutcliffe and L. Macaulay. Cambridge University Press, pp. 443 – 460 (cit. on p. 107).

Gudgin, Martin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon (Apr. 2007). *SOAP Version 1.2*. W3C Recommendation. URL: http://www.w3.org/TR/soap12. W3C (cit. on p. 97).

Hager, Henning, Stefan Hennig, Marc Seißler, and Annerose Braune (2011). "Modelltransformationen in nutzer-zentrierten Entwurfsprozessen der Automation". In: *i-com* 10.3, pp. 19 – 25 (cit. on pp. 134, 135).

Heidenreich, Florian, Jan Kopcsek, and Uwe Aßmann (2010). "Safe Composition of Transformations". In: *Theory and Practice of Model Transformations*. Ed. by Laurence Tratt and Martin Gogolla. Vol. 6142. Lecture Notes in Computer Science. Springer, pp. 108 – 122 (cit. on p. 146).

Heidenreich, Florian, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende (2009). "Derivation and Refinement of Textual Syntax for Models". In: *Model Driven Architecture - Foundations and Applications*. Ed. by Richard Paige, Alan Hartman, and Arend Rensink. Vol. 5562. Lecture Notes in Computer Science. Springer, pp. 114 – 129 (cit. on p. 104).

Helms, James, Robbie Schaefer, Kris Luyten, Jo Vermeulen, Marc Abrams, Adrien Coyette, and Jean Vanderdonckt (2009). "Human-Centered Engineering Of Interactive Systems With The User Interface Markup Language". In: *Human-Centered Software Engineering*. Ed. by Ahmed Seffah, Jean Vanderdonckt, and Michel C. Desmarais. Springer, pp. 139–171 (cit. on p. 11).

Hennig, Stefan, Jan Van den Bergh, Kris Luyten, and Annerose Braune (2011). "User Driven Evolution of User Interface Models — The FLEPR Approach". In: *Human-Computer Interaction – INTERACT 2011*. Ed. by Pedro Campos, Nicholas Graham, Joaquim Jorge, Nuno Nunes, Philippe Palanque, and Marco Winckler. Vol. 6948. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, pp. 610–627 (cit. on p. 135).

Hovsepyan, Aram, Stefan Van Baelen, Koen Yskout, Yolande Berbers, and Wouter Joosen (2007). "Composing Application Models and Security Models: on the Value of Aspect-Oriented Technologies". In: *Proceedings of the Eleventh International*

Workshop on Aspect-Oriented Modeling AOM@MODELS 2007, pp. 1 – 10 (cit. on p. 147).

ICONICS (2011). *HMI and SCADA Solutions*. Website, last visited: September 23rd, 2011. URL: http://iconics.com/Home/Products/HMI-and-SCADA.aspx (cit. on p. 10).

ISO/IEC (2006). *Information technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation — Schematron*. ISO/IEC 19757-3: http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html (cit. on p. 25).

Jardim Nunes, Nuno and João Falcão Cunha (2000). "Towards a UML Profile for Interaction Design: The Wisdom Approach". In: *«UML» 2000 — The Unified Modeling Language*. Ed. by Andy Evans, Stuart Kent, and Bran Selic. Vol. 1939. Lecture Notes in Computer Science. Springer, pp. 101 – 116 (cit. on p. 22).

Johannsen, Gunnar (1993). *Mensch-Maschine-Systeme*. Springer (cit. on pp. 1, 5, 108, 118).

Kelly, Steven and Risto Pohjonen (2009). "Worst Practices for Domain-Specific Modeling". In: *IEEE Software* 26.4, 22 — 29 (cit. on pp. 30, 31, 80, 130).

Kleppe, Anneke (2006). "MCC: A Model Transformation Environment". In: *Model Driven Architecture – Foundations and Applications*. Ed. by Arend Rensink and Jos Warmer. Vol. 4066. Lecture Notes in Computer Science. Springer, pp. 173 – 187 (cit. on p. 146).

– (2009). *Software Language Engineering — Creating Domain-Specific Languages using Metamodels*. 1. ed. Addison-Wesley (cit. on pp. 17, 19, 69, 79, 94).

Koch, Nora, Alexander Knapp, Gefei Zhang, and Hubert Baumeister (2008). "UML-Based Web Engineering". In: *Web Engineering: Modelling and Implementing Web Applications*. Ed. by Gustavo Rossi, Oscar Pastor, Daniel Schwabe, and Luis Olsina. Human-Computer Interaction Series. Springer, pp. 157 – 191 (cit. on p. 22).

Kolovos, Dimitrios S., Richard F. Paige, and Fiona A.C. Polack (2009). "The Grand Challenge of Scalability for Model Driven Engineering". In: *Models in Software Engineering*. Ed. by Michel Chaudron. Vol. 5421. Lecture Notes in Computer Science. Springer, pp. 48 – 53 (cit. on pp. 93, 104).

Kolovos, Dimitrios S., Richard F. Paige, Fiona A.C. Polack, and Louis M. Rose (2007). "Update Transformations in the Small with the Epsilon Wizard Language". In: *Journal of Object Technology* (cit. on p. 138).

Lange, Jürgen, Frank Iwanitz, and Thomas J. Burke (2010). *OPC: Von Data Access bis Unified Architecture*. 4. Auflage. VDE Verlag (cit. on pp. 32, 45, 71).

Lego (2011). *Mindstorms*. Website, last visited: October 26th, 2011. URL: http://mindstorms.lego.com/en-us/default.aspx (cit. on p. 118).

Limbourg, Quentin, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero (2005). "USIXML: A Language Supporting Multi-path Development of User Interfaces". In: *Engineering Human Computer Interaction and Interactive Systems*. Ed. by Rémi Bastide, Philippe Palanque, and Jörg Roth. Lecture Notes in Computer Science. Springer (cit. on pp. 12, 13).

Lunze, Jan (2008). *Automatisierungstechnik: Methoden für die Überwachung und Steuerung kontinuierlicher und ereignisdiskreter Systeme*. 2. überarbeitete Auflage. Oldenbourg Verlag (cit. on pp. 1, 108, 120).

Mahnke, Wolfgang, Stefan-Helmut Leitner, and Matthias Damm (2009). *OPC Unified Architecture*. Springer (cit. on p. 32).

Martinie, Célia and Philippe Palanque (2011). "A Multi-Models Based Development Process for Critical Interactive Systems Integrating Formal and Informal Approaches". In: *Proceedings of the First International Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeisMoto), organized at INTERACT 2011 – 13th IFIP TC13 Conference on Human-Computer-Interaction* (cit. on pp. 25, 28, 144).

Meixner, Gerrit (2010). "Entwicklung einer modellbasierten Architektur für multimodale Benutzungsschnittstellen". PhD Thesis. TU Kaiserslautern (cit. on pp. 133, 134).

Meixner, Gerrit, Marc Seissler, and Kai Breiner (2011). "Model-Driven Useware Engineering". In: *Model-Driven Development of Advanced User Interfaces*. Ed. by Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke. Vol. 340. Studies in Computational Intelligence. Springer, pp. 1 – 26 (cit. on pp. 133, 134).

Mellor, Stephen J. and Marc J. Balcer (2002). *Executable UML — A Foundation for Model-Driven Architecture*. Addison-Wesley (cit. on pp. 14, 40, 41).

Mellor, Stephen J., Kendall Scott, Axel Uhl, and Dirk Weise (2004). *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley (cit. on pp. 14, 16).

Mens, Tom, Krzysztof Czarnecki, and Pieter Van Gorp (2006). "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152, pp. 125 – 142 (cit. on pp. 15, 94, 95).

Mentor Graphics (2008). *Object Action Language Reference Manual*. Company Specification (cit. on p. 81).

– (2011). *BridgePoint*. Website, last visited: November 22nd, 2011. URL: http://www.mentor.com/products/sm/model_development/bridgepoint/ (cit. on p. 81).

Menzel, Guido, Dieter Feier, Gerhard Adam, and Thomas Hauff (2003). "Investitionssicherheit von Einrichtungen der Prozessleittechnik – eine (un)lösbare Aufgabe?" In: *atp* Heft 3, pp. 26 – 30 (cit. on pp. 3, 4).

Mernik, Marjan, Jan Heering, and Anthony M. Sloane (2005). "When and how to develop domain-specific languages". In: *ACM Computing Surveys (CSUR)* 37 (4), pp. 316 – 344 (cit. on p. 21).

Meskens, Jan, Kris Luyten, and Karin Coninx (2010). "D-Macs: Building Multi-Device User Interfaces by Demonstrating, Sharing and Replaying Design Actions". In: *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. UIST '10, 129 — 138 (cit. on p. 144).

Meskens, Jan, Mieke Haesen, Kris Luyten, and Karin Coninx (2009). "User-Centered Adaptation of User Interfaces for Heterogeneous Environments". In: *Advances in Semantic Media Adaptation and Personalization, Volume 2*. Auerbach Publications, pp. 43 – 65 (cit. on pp. 27, 98).

MetaCase (2011). *Domain Specific Modeling with MetaEdit+*. Website, last visited: September 30rd, 2011. URL: http://www.metacase.com/ (cit. on p. 20).

Michotte, Benjamin and Jean Vanderdonckt (2008). "GrafiXML, a Multi-target User Interface Builder Based on UsiXML". In: *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*. IEEE Computer Society, pp. 15 – 22 (cit. on p. 24).

Microsoft (2011). *Extensible Application Markup Language (XAML)*. Website, last visited: September 27th, 2011. URL: http://www.microsoft.com/download/en/details.aspx?displayLang=en&id=19600 (cit. on p. 14).

Miller, Joaquin and Jishnu Mukerji (2003). *MDA Guide Version 1.0.1*. Tech. rep. OMG (cit. on p. 16).

Mohagheghi, Parastoo, Miguel A. Fernandez, Juan A. Martell, Mathias Fritzsche, and Wasif Gilani (2009). "MDE Adoption in Industry: Challenges and Success Criteria". In: *Models in Software Engineering*. Ed. by Michel Chaudron. Vol. 5421. Lecture Notes in Computer Science. Springer, pp. 54 – 59 (cit. on p. 17).

Moody, Daniel (2009). "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35 (6), pp. 756 – 779 (cit. on p. 24).

Myers, Brad, Scott E. Hudson, and Randy Pausch (2000). "Past, Present, and Future of User Interface Software Tools". In: *ACM Transactions on Computer-Human Interaction* Volume 7 (Issue 1), pp. 3–28 (cit. on pp. 24, 144).

NAMUR (2005). *Alarm Management*. NAMUR NA 102. URL: http://www.namur.de (cit. on pp. 49, 74).

– (2006). *IT-Security for Industrial Automation Systems: Constraints for measures applied in process industries*. NAMUR NA 115. URL: http://www.namur.de (cit. on p. 146).

Nichols, Jeffrey, Duen Horng Chau, and Brad A. Myers (2007). "Demonstrating the Viability of Automatically Generated User Interfaces". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '07, pp. 1283 – 1292 (cit. on pp. 4, 26).

Nichols, Jeffrey and Brad A. Myers (2009). "Creating a lightweight user interface description language: An overview and analysis of the personal universal controller project". In: *ACM Transactions on Computer-Human Interaction* 16 (4), 17:1 – 17:37 (cit. on p. 11).

No Magic (2011). *MagicDraw UML*. Website, last visited: September 29rd, 2011. URL: http://www.magicdraw.com (cit. on p. 20).

Norman, Donald A. and Stephen W. Draper (1986). *User Centered System Design: New Perspectives on Human-computer Interaction* (cit. on p. 131).

OHP (2012). *ProWin*. Website, last visited: January 13th, 2012. URL: http://www.ohp.de/produkte-prowin.htm (cit. on p. 10).

OMF (2010). *Action Language For Foundational UML (ALF) 1.0 — Beta 1*. OMG Available Specification (cit. on p. 81).

OMG (2004). *UML Human-Usable Textual Notation, Version 1.0*. OMG Available Specification (cit. on p. 20).

– (2008). *Semantics of a Foundational Subset for Executable UML Models*. OMG Available Specification (cit. on pp. 40, 41, 63, 114).

OMG (2010). *Meta Object Facility, v2.4 — Beta 2*. OMG Available Specification (cit. on pp. 16, 18).

OMG (2010). *OMG Unified Modeling Language™(OMG UML), Superstructure, Version 2.4*. OMG Available Specification (cit. on pp. 40, 41, 43, 51, 63, 82).

OMG (2010a). *Object Management Group Website*. Website, last visited: September 23rd, 2011. URL: http://www.omg.org/ (cit. on p. 16).

– (2010b). *UML® Resource Page*. Website, last visited: September 23rd, 2011. URL: http://www.uml.org/ (cit. on p. 16).

– (2011a). *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1*. OMG Available Specification (cit. on p. 16).

– (2011b). *Object Constraint Language, v2.3 — Beta 2*. OMG Available Specification (cit. on p. 19).

OPC Foundation (2004). *OPC XML-DA Specification, Version 1.01*. OPC Foundation Available Industry Standard Specification (cit. on pp. 71, 77).

Olsen Jr., Dan R. (2007). "Evaluating User Interface Systems Research". In: *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*. UIST '07. Newport, Rhode Island, USA: ACM, 251–258 (cit. on pp. 107, 126–130, 139).

Ottensooser, Avner, Alan Fekete, Hajo A. Reijers, Jan Mendling, and Con Menictas (2012). "Making sense of business process descriptions: An experimental comparison of graphical and textual notations". In: *Journal of Systems and Software* 85.3, pp. 596 – 606 (cit. on pp. 18, 20, 80).

Paternò, Fabio (2000). *Model-Based Design and Evaluation of Interactive Applications*. Applied Computing. Springer (cit. on pp. 11, 14).

Paternò, Fabio, Carmen Santoro, and Lucio Davide Spano (2009). "MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* Volume 16.Issue 4 (cit. on pp. 13, 23, 24).

Petrasch, Roland and Oliver Meimberg (2006). *Model Driven Architecture — eine praxisorientierte Einführung in die MDA*. Heidelberg: dpunkt-Verl. (cit. on p. 15).

Pietrek, Georg, Jens Trompeter, Juan Carlos Flores Beltran, Boris Holzer, Thorsten Kamann, Michael Kloss, Steffen A. Mork, Benedikt Niehues, and Karsten Thoms (2007). *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Ed. by Georg Pietrek and Jens Trompeter. Entwickler.press (cit. on p. 16).

Puerta, A. R (Aug. 1997). "A model-based interface development environment". In: *Software, IEEE* 14.4, pp. 40–47 (cit. on p. 11).

Puerta, Angel and Jacob Eisenstein (2001). "XIML: A Universal Language for User Interfaces". In: *RedWhale Software*. URL: http://www.ximl.org (cit. on p. 11).

Raneburger, David, Roman Popp, Sevan Kavaldjian, Hermann Kaindl, and Jürgen Falb (2011a). "Optimized GUI Generation for Small Screens". In: *Model-Driven Development of Advanced User Interfaces*. Ed. by Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke. Vol. 340. Studies in Computational Intelligence. Springer, pp. 107 – 122 (cit. on p. 147).

Raneburger, David, Alexander Schörkhuber, Hermann Kaindl, and Jürgen Falb (2011b). "UI Development Support through Model-integrity Checks in a Discourse-based Generation Framework". In: The First International Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeisMoto), official IFIP WG 2.7/13.4 Workshop of INTERACT 2011 — 13th IFIP TC13 Conference on Human-Computer Interaction (cit. on pp. 26, 93).

Roscher, Dirk, Grzegorz Lehmann, Veit Schwartze, Marco Blumendorf, and Sahin Albayrak (2011). "Dynamic Distribution and Layouting of Model-Based User Interfaces in Smart Environments". In: *Model-Driven Development of Advanced User Interfaces*. Ed. by Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke. Vol. 340. Studies in Computational Intelligence. Springer, pp. 171 – 197 (cit. on pp. 147, 148).

Rose, Louis M., Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack (2008). "The Epsilon Generation Language". In: *Model Driven Architecture – Foundations and Applications*. Vol. 5095. Lecture Notes in Computer Science. Springer (cit. on pp. 16, 96, 102).

Ruscio, Davide Di, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio (Apr. 2006). *Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs*. Tech. rep. n. 06.02. Laboratoire d'Informatique de Nantes-Atlantique (LINA). URL: http://hal.ccsd.cnrs.fr/ccsd-00023008/en (cit. on p. 17).

Scheidgen, Markus (2008). "Textual Modelling Embedded into Graphical Modelling". In: *Model Driven Architecture – Foundations and Applications*. Ed. by Ina Schieferdecker and Alan Hartman. Vol. 5095. Lecture Notes in Computer Science. Springer, pp. 153 – 168 (cit. on p. 104).

Schmitz, Stefan and Ulrich Epple (2007). "Automatisierte Projektierung von HMI-Oberflächen". In: *VDI-Berichte 1980*. Baden-Baden (cit. on p. 10).

Schäfer, Robbie (2007). "Model-Based Development of Multimodal and Multi-Device User Interfaces in Context-Aware Environments". PhD Thesis. Universität Paderborn (cit. on pp. 23, 133).

Seißler, Marc, Kai Breiner, and Gerrit Meixner (2011). "Towards Pattern-Driven Engineering of Run-Time Adaptive User Interfaces for Smart Production Environments". In: *Human-Computer Interaction. Design and Development Approaches*. Ed. by Julie Jacko. Vol. 6761. Lecture Notes in Computer Science. Springer, pp. 299 – 308 (cit. on p. 148).

Selic, Bran (2003). "The Pragmatics of Model-Driven Development". In: *IEEE Software* 20.5, 19 — 25 (cit. on p. 42).

Sheridan, Thomas B. (1992). *Telerobotics, Automation, and Human Supervisory Control*. MIT Press (cit. on pp. 1, 2).

– (2000). "HCI in Supervisory Control: Twelve Dilemmas". In: *Human error and system design and management*. Ed. by P. Elzer, R. Kluwe, and B. Boussoffara. Vol. 253. Lecture Notes in Control and Information Sciences. Springer, pp. 1 – 12 (cit. on p. 28).

Siemens (2011). *SCADA System SIMATIC WinCC*. Website, last visited: September 23rd, 2011. URL: http://www.siemens.com/wincc (cit. on p. 10).

Silingas, Darius, Ruslanas Vitiutinas, Andrius Armonas, and Lina Nemuraite (2009). "DOMAIN-SPECIFIC MODELING ENVIRONMENT BASED ON UML PROFILES". In: *No Magic Training and Information Portal*. Website last visited: September 29th, 2011. URL: http://training.nomagic.com/index.php?option=com_content&view=article&id=327&Itemid=66 (cit. on p. 20).

Stahl, Thomas, Markus Völter, Sven Efftinge, and Arno Haase (2007). *Modellgetriebene Softwareentwicklung — Techniken, Engineering, Management*. 2., aktualisierte und erweiterte Auflage. Heidelberg: dpunkt.verlag GmbH (cit. on pp. 14, 15, 17, 21, 24, 30, 39, 80, 88, 89, 94, 96, 130).

Starr, Leon (2002). *Executable UML: How to build Class Models*. Prentice Hall PTR (cit. on p. 41).

– (2003). *Scrall: Starr's Concise Relational Action Language*. Model Integration, LLC. URL: http://www.modelint.com (cit. on pp. 81–83, 89).

Steinberg, Dave, Frank Budinsky, Marcelo Paternostro, and Ed Merks (2008). *EMF Eclipse Modeling Framework*. 2. edition. Addison-Wesley (cit. on pp. 16, 18).

Streitferdt, Detlef, Georg Wendt, Philipp Nenninger, Alexander Nyssen, and Horst Lichter (2008). "Model Driven Development Challenges in the Automation Domain". In: *32nd Annual IEEE International Computer Software and Applications Conference, 2008. COMPSAC '08.* Pp. 1372 – 1375 (cit. on pp. 24, 26).

Strembeck, Mark and Uwe Zdun (2009). "An approach for the systematic development of domain-specific languages". In: *Software: Practice and Experience* 39.15, pp. 1253 – 1292 (cit. on pp. 17, 18, 20, 21, 24, 62, 88).

Sun, Yu, Zekai Demirezen, Tomaž Lukman, Marjan Mernik, and Jeff Gray (2008). "Model Transformations Require Formal Semantics". In: *Domain-Specific Program Development*. Ed. by Julia Lawall and Laurent Réveillère. Nashville, United States, p. 5 (cit. on pp. 79, 102).

TOPCASED (2011). *TOPCASED: The Open-Source Toolkit for Critical Systems*. Website, last visited: November 17th, 2011. URL: http://www.topcased.org (cit. on p. 91).

Tarjan, Robert (1972). "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2, pp. 146 – 160 (cit. on p. 96).

Tauchnitz, Thomas and Uwe Maier (2004). "Prozessleitebene (PLS)". In: *Handbuch der Prozessautomatisierung*. Ed. by Karl Friedrich Früh and Uwe Maier. 3. Auflage. Oldenburg Industrieverlag München, pp. 216–228 (cit. on p. 33).

Teurich-Wagner, Sören (2004). "MDA: Weg oder Irrweg". In: *Modellierung 2004*. Vol. P-45. GI, Gesellschaft für Informatik, Bonn, pp. 223 – 227 (cit. on p. 17).

Thompson, Henry S., Noah Mendelsohn, David Beech, Murray Maloney, Shudi (Sandy) Gao, and C. M. Sperberg-McQueen (2011). *W3C XML Schema Definition Language (XSD) 1.1*. URL: http://www.w3.org/TR/2011/CR-xmlschema11-1-20110721/ (cit. on p. 18).

Urbas, Leon and Falk Doherr (2011). "autoHMI: a model driven software engineering approach for HMIs in process industries". In: *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, pp. 627 – 631 (cit. on pp. 8, 9, 131, 132).

Urbas, Leon, Stefan Hennig, Henning Hager, Falk Doherr, and Annerose Braune (2011). "Towards context adaptive HMIs in process industries". In: *Proceedings of the 9th IEEE International Conference on Industrial Informatics* (cit. on p. 132).

VDI/VDE (1997 – 2005). *Process control using display screens*. VDI/VDE 3699 Guideline (cit. on pp. 78, 133, 136, 148).

– (1997). *Process control using display screens, Part 4: Curves*. VDI/VDE 3699 Guideline (cit. on p. 51).

– (1998). *Process control using display screens, Part 5: Messages*. VDI/VDE 3699 Guideline (cit. on pp. 48, 49, 51).

VDI/VDE (1999). *Process control using display screens, Part 3: Mimics*. VDI/VDE 3699 Guideline (cit. on pp. 34, 35, 47, 51, 52, 54, 57, 58, 60, 112).

– (2002). *Process control using display screens, Part 6: Interaction procedures and devices*. VDI/VDE 3699 Guideline (cit. on pp. 50, 56, 112).

– (2005). *Process control using display screens, Part 2: Prinicples*. VDI/VDE 3699 Guideline (cit. on pp. 3, 28, 47, 52, 53, 56).

– (2007). *IT-security for industrial automation — General model*. VDI/VDE 2182 Guideline (cit. on p. 146).

– (2011). *Building automation and control systems (BACS), Part 7: Design of user interfaces*. VDI/VDE 3814 Guideline (cit. on p. 51).

Van den Bergh, Jan and Karin Coninx (2005). "Using UML 2.0 and Profiles for Modeling Context-Sensitive User Interfaces". In: *Proceedings of the MoDELS'05 Workshop on Model Driven Development of Advanced User Interfaces*. Ed. by Andreas Pleuss, Jan Van den Bergh, Heinrich Hußmann, and Stefan Sauer (cit. on p. 22).

Van den Bergh, Jan, Kris Luyten, and Karin Coninx (2011). "CAP3: Context-Sensitive Abstract User Interface Specification". In: *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '11. New York, NY, USA: ACM, 31 — 40 (cit. on p. 136).

Vanderdonckt, Jean (2005). "A MDA-Compliant Environment for Developing User Interfaces of Information Systems". In: *Advanced Information Systems Engineering*, pp. 16–31 (cit. on pp. 12, 21).

– (2008). "Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures". In: *Proceedings of 5th Annual Romanian Conference on Human-Computer Interaction ROCHI'2008*. Ed. by S. Buraga and I. Juvina (cit. on p. 26).

Vanhooff, Bert, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers (2007). "UniTI: A Unified Transformation Infrastructure". In: *MoDELS*, pp. 31 – 45 (cit. on p. 146).

Völter, Markus (2009). "MD* Best Practices". In: *Journal of Object Technology* 8.6, pp. 79 – 102. URL: http://dblp.uni-trier.de/db/journals/jot/jot8.html#Volter09 (cit. on pp. 36, 39, 60, 79, 80, 88, 89, 92–94, 96, 130).

Wagelaar, Dennis and Viviane Jonckers (2005). "Explicit Platform Models for MDA". In: *Model Driven Engineering Languages and Systems*. Ed. by Lionel Briand and Clay Williams. Lecture Notes in Computer Science. Springer, pp. 367–381 (cit. on p. 13).

Wagelaar, Dennis, Massimo Tisi, Jordi Cabot, and Frédéric Jouault (2011). "Towards a General Composition Semantics for Rule-Based Model Transformation". In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark, and Thomas Kühne. Vol. 6981. Lecture Notes in Computer Science. Springer, pp. 623 – 637 (cit. on p. 146).

Wilkie, Ian, Adrian King, Mike Clarke, Chas Weaver, Chris Raistrick, and Paul Francis (2003). *UML ASL Reference Guide: ASL Language Level 2.5*. Company Specification (cit. on p. 81).

Wonderware (2011). *HMI/SCADA Software Solutions*. Website, last visited: September 23rd, 2011. URL: http://global.wonderware.com/EN/Pages/WonderwareHMISCADA.aspx (cit. on p. 10).

Zacher, Serge and Claude Wolmering (2009). *Prozessvisualisierung: Methoden, Programme, Projekte für die Regelung und Steuerung mit SPS*. Zacher (cit. on pp. 8, 9, 32, 49).

Zeppenfeld, Klaus and Regine Wolters (2006). *Generative Software-Entwicklung mit der MDA*. 1. Auflage. Heidelberg: Elsevier Spektrum Akademischer Verlag (cit. on p. 14).

Zühlke, Detlef (2004). *Useware-Engineering für technische Systeme*. VDI-Buch. Springer (cit. on pp. 28, 131, 133).

atvise (2011). *atvise® — The HMI & SCADA Revolution*. Website, last visited: September 23rd, 2011. URL: http://www.atvise.com/ (cit. on p. 10).

da Silva, Paulo Pinheiro and Norman W. Paton (2003). "User Interface Modeling in UMLi". In: *IEEE Software* 20 (4), pp. 62 – 69 (cit. on p. 22).

oAW (2011). *openArchitectureWare Check/Xtend/Xpand Reference*. Website, last visited: September 28rd, 2011. URL: http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html (cit. on pp. 19, 93, 96, 102).

Appendix

Appendix A

Sample Visualization Applications

This chapter presents screenshots of selected sample visualization solutions. They have been captured from real projects and can be used in this thesis with the permission of *Gruppe 4.5.4 Gebäudeautomation, Technische Universität Dresden* (Section A.1) and *C.E.P Anlagenautomatisierung Dresden GmbH* (Section A.2 and Section A.3).

A.1 Power Supply Network Monitoring in the Technische Universität Dresden

All utilities of the buildings of the *Technische Universität Dresden* are monitored from a centralized control room. This visualization solution allows among others for monitoring of climate conditions and the states of the (waste-)water and power supply networks. Figure A.1 and Figure A.2 show two different power supply network monitoring mimics residing at different hierarchy levels.

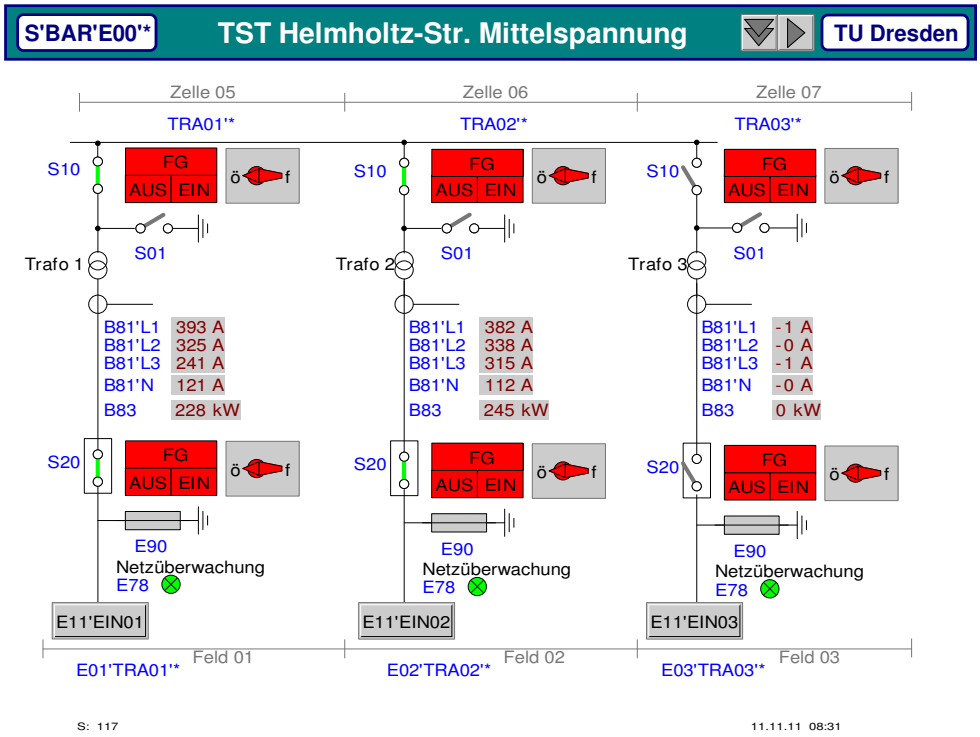


Figure A.1: Mimic of a transformer station in which the power is transformed from medium voltage (from the electricity supplier) to supply voltage (to the consumer). Characteristic in this mimic are the *three stage switches*: Only after releasing this switch locally, a second release must happen through the visualization solutions before it can be operated.

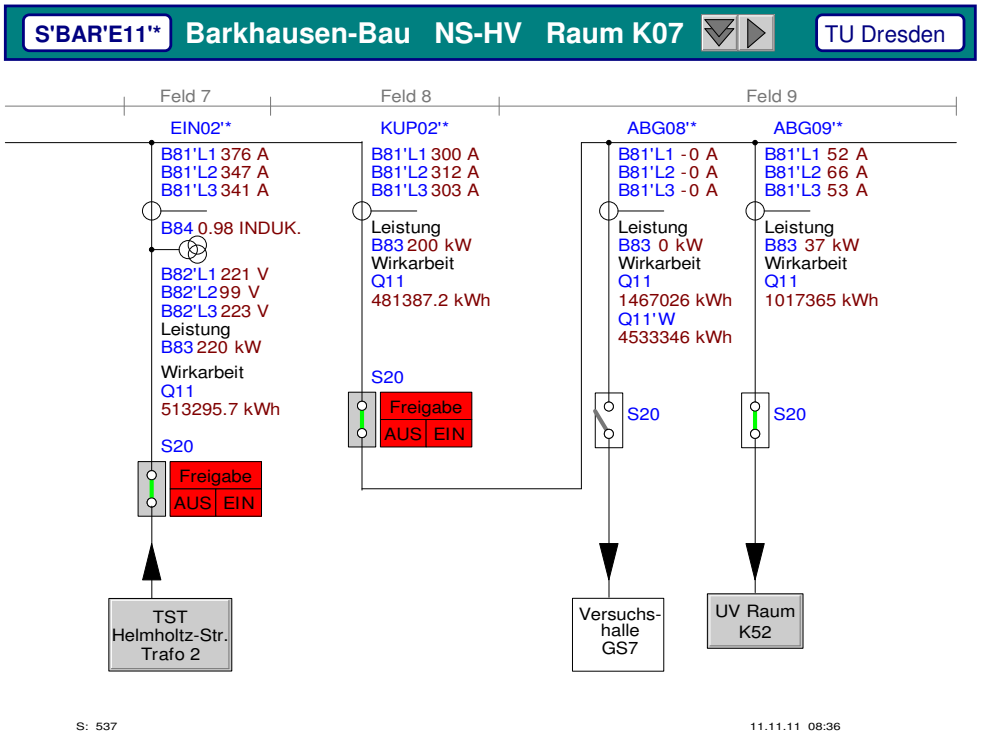


Figure A.2: Mimic of the subsequent hierarchy level: Operative states of the network inside the building can be monitored.

A.2 Process Industries

The process industry is characterized by technical processes that are operated continuously or in a batch manner. Figure A.3 presents an excerpt of a visualization solution of a continuous pharmaceutical process.

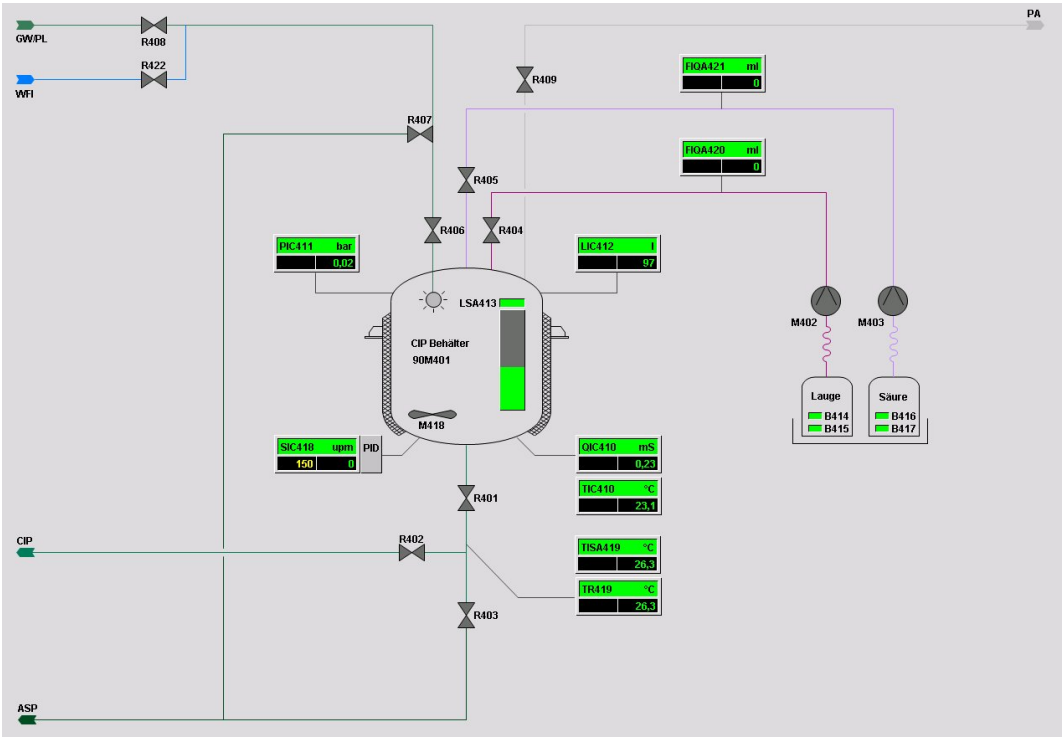


Figure A.3: Mimic showing a part of a pharmaceutical process: It is characterized by a static background image augmented with dynamic elements. These dynamic elements reflect the current state of the process by showing actual process values numerically or by means of elements that vary their height (size animations).

A.3 Factory Automation

Factory automation processes are characterized by discrete processes using machines or robots. Figure A.4 presents an excerpt of a wafer production process.

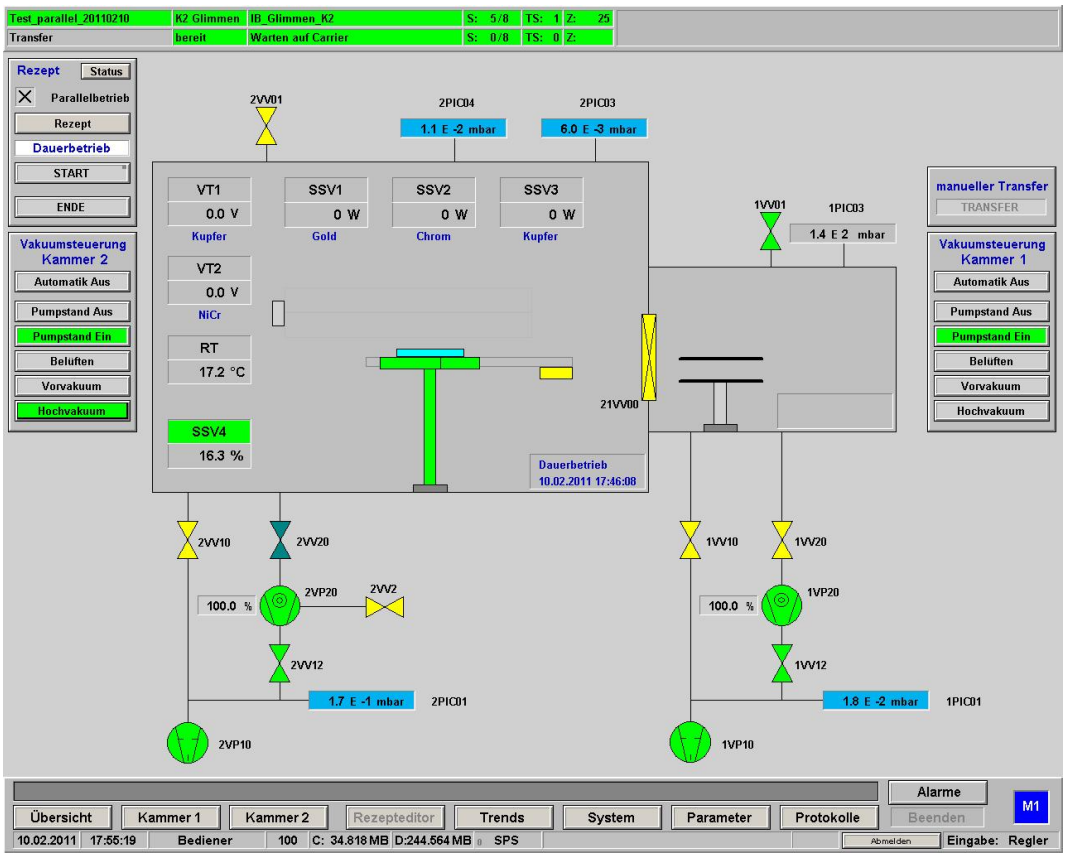


Figure A.4: Mimic for monitoring a wafer production line: A robot arm is shown in the center. Hence, user interfaces for monitoring and operating of factory automation processes are characterized by moving elements (position animations).

Appendix B

Core Language Model

This chapter presents the entire and complete *Core Language Model* of the domain specific language MOVISA.

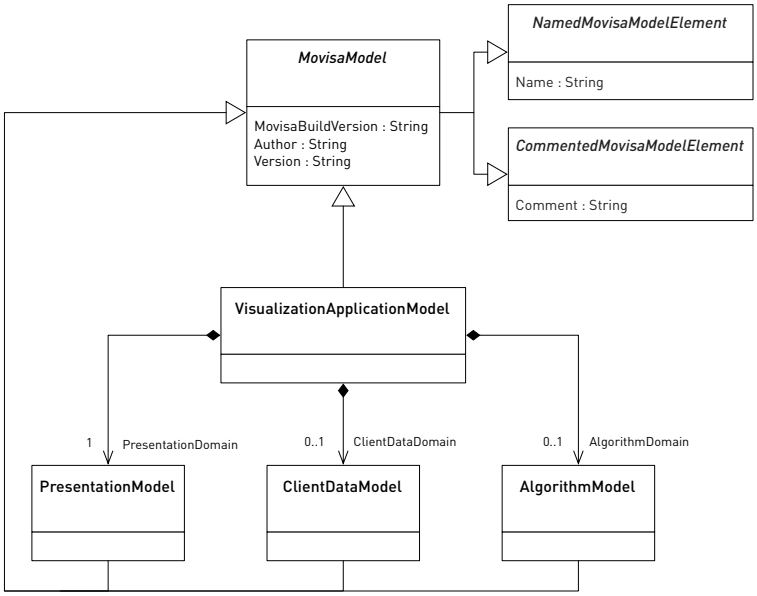


Figure B.1: Core Language Model: Movisa Root.

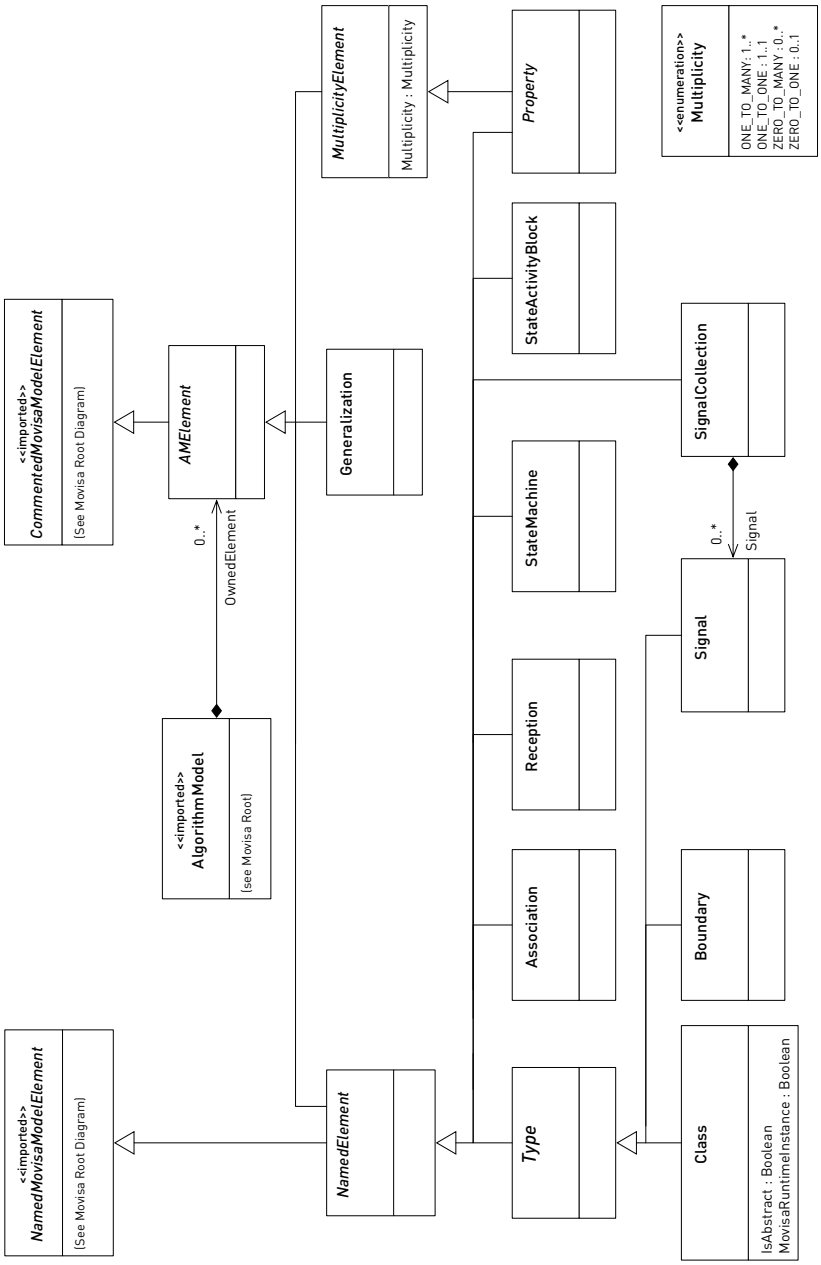


Figure B.2: Core Language Model: Algorithm Root.

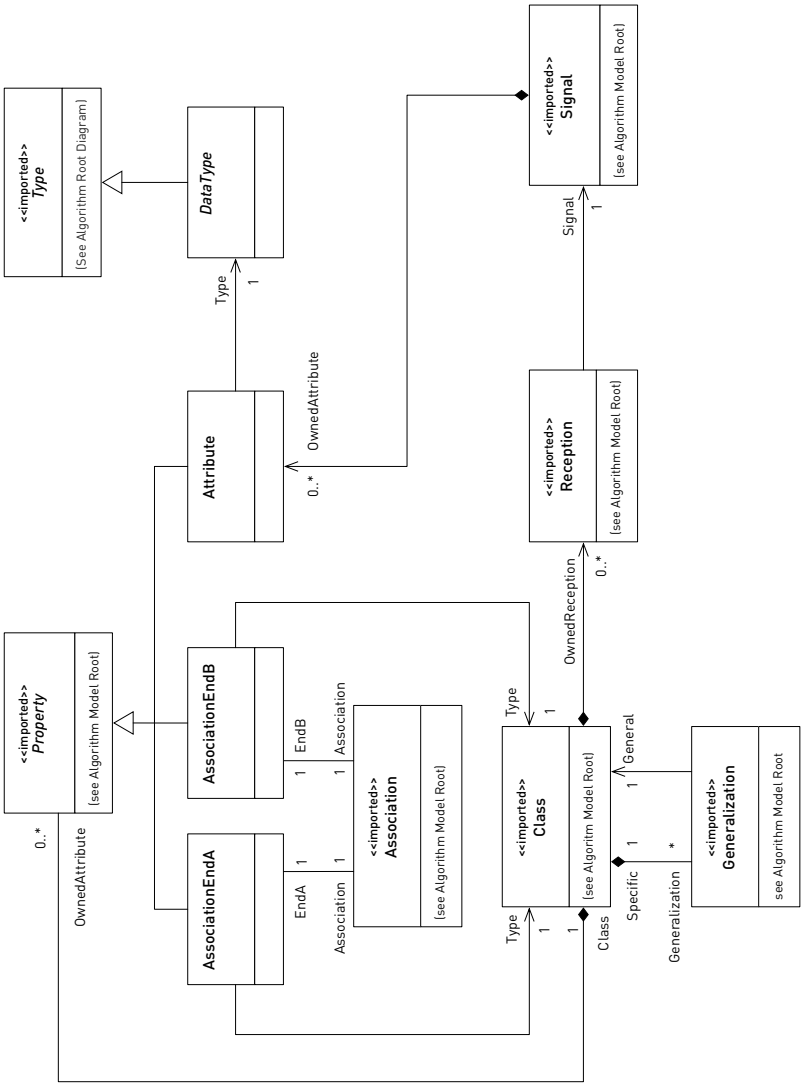


Figure B.3: Core Language Model: Class Subsystem.

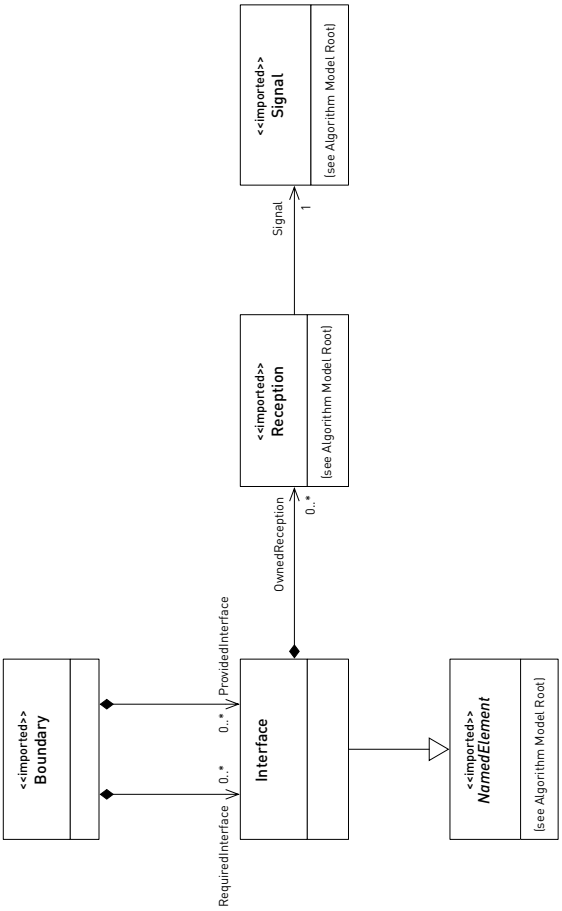


Figure B.4: Core Language Model: Boundary Subsystem.

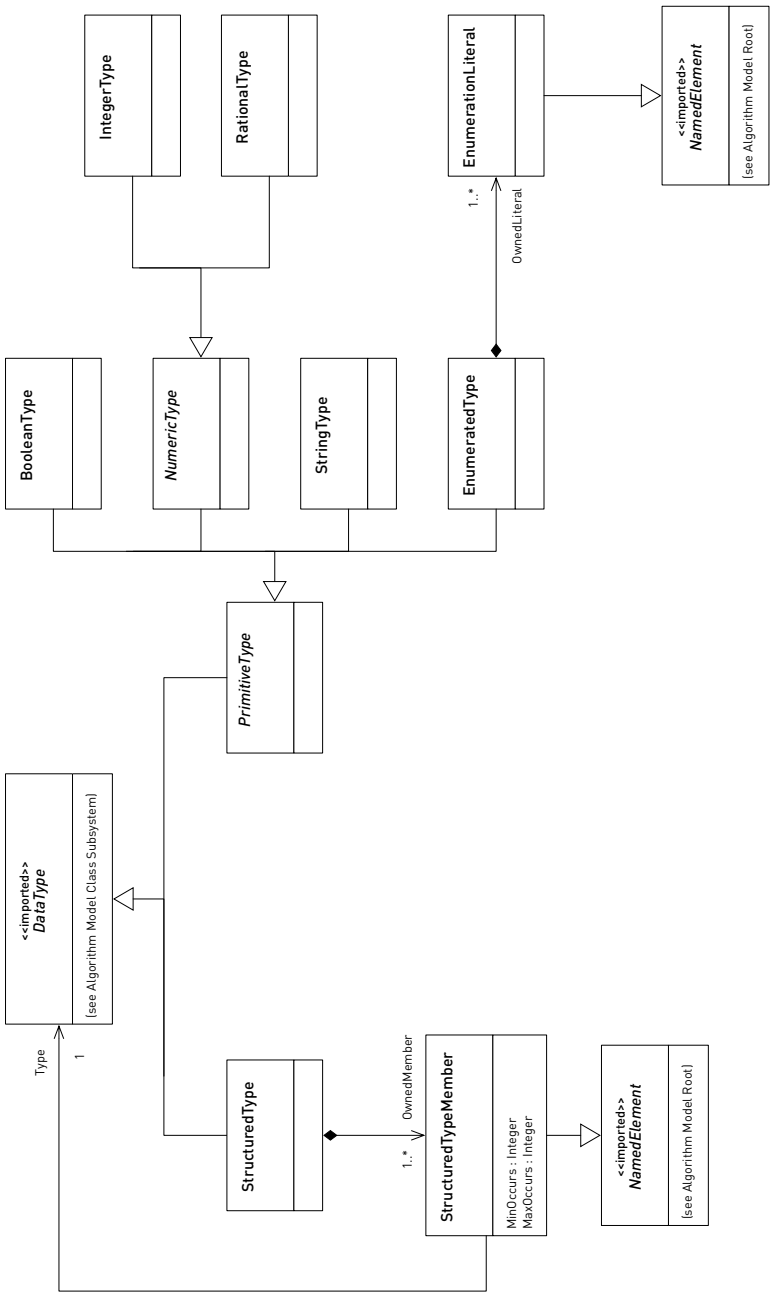


Figure B.5: Core Language Model: Data Type Subsystem.

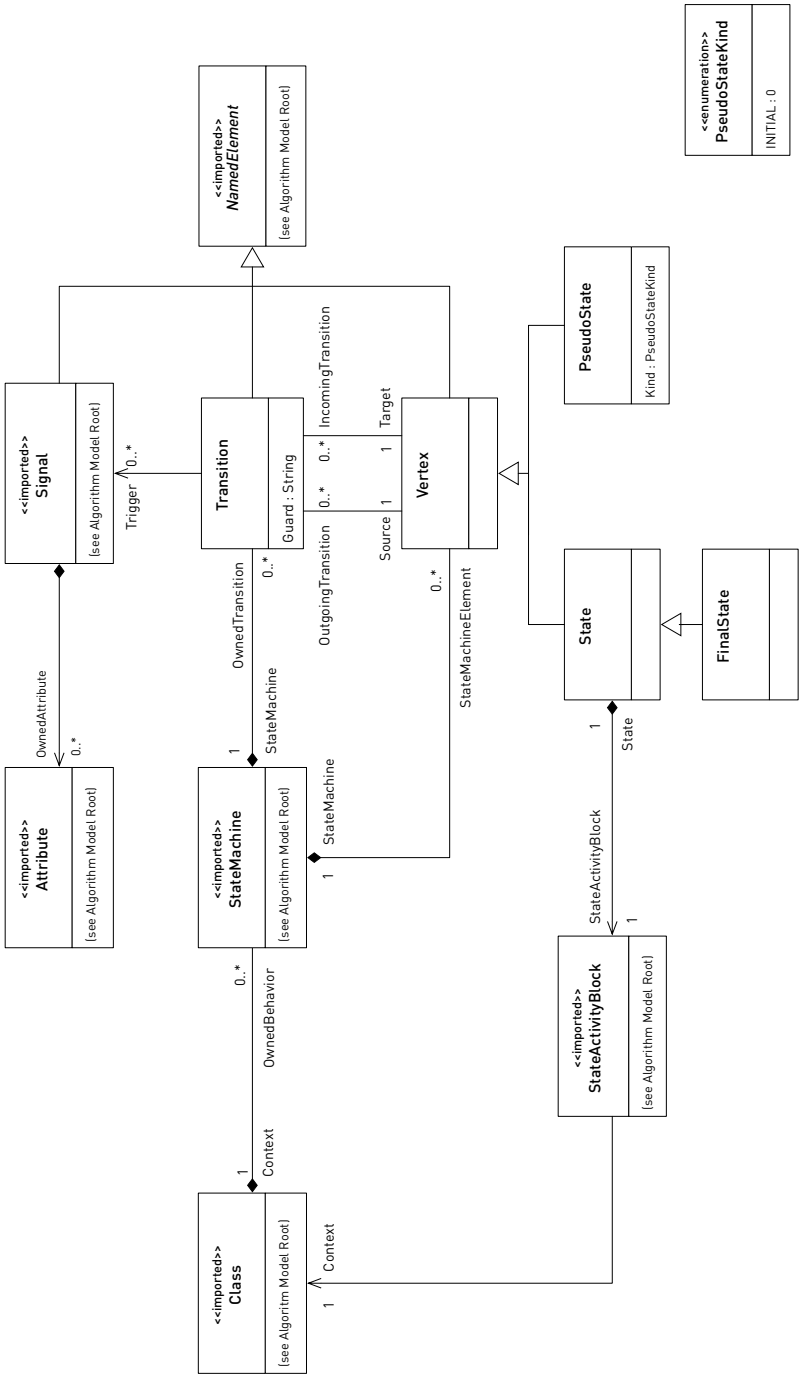


Figure B.6: Core Language Model: State Machine Subsystem.

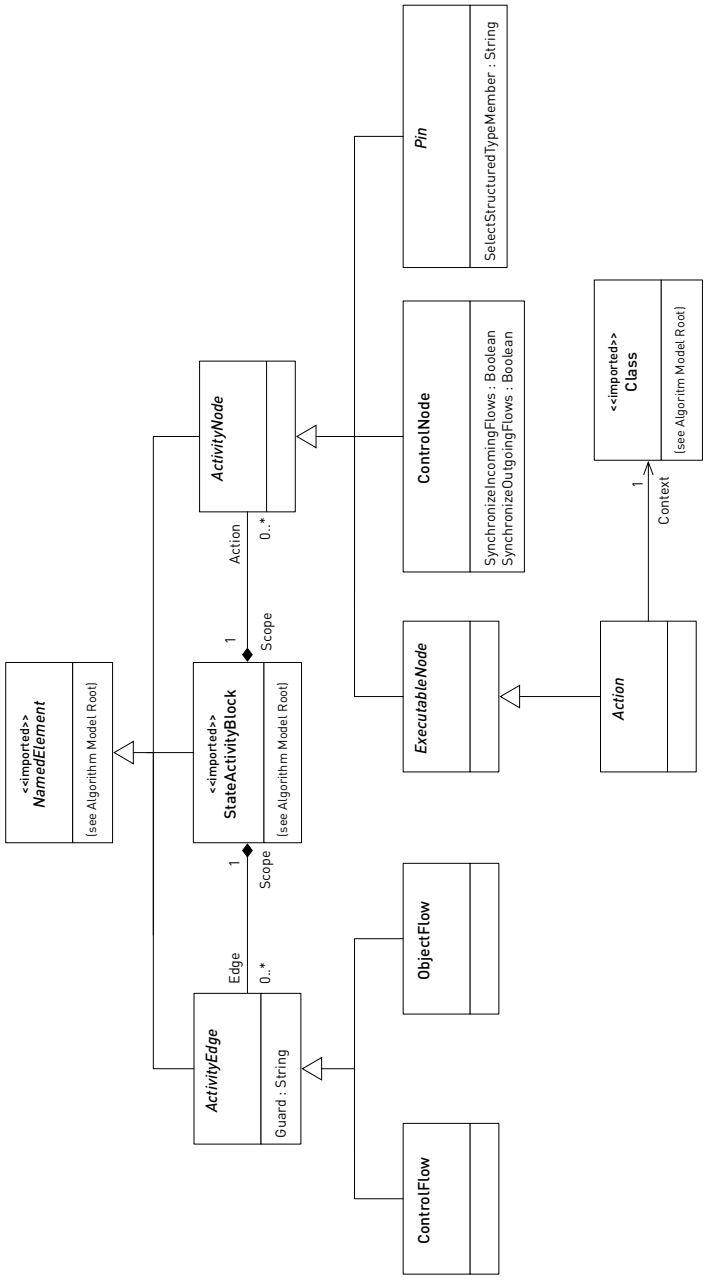


Figure B.7: Core Language Model: Action Root Subsystem.

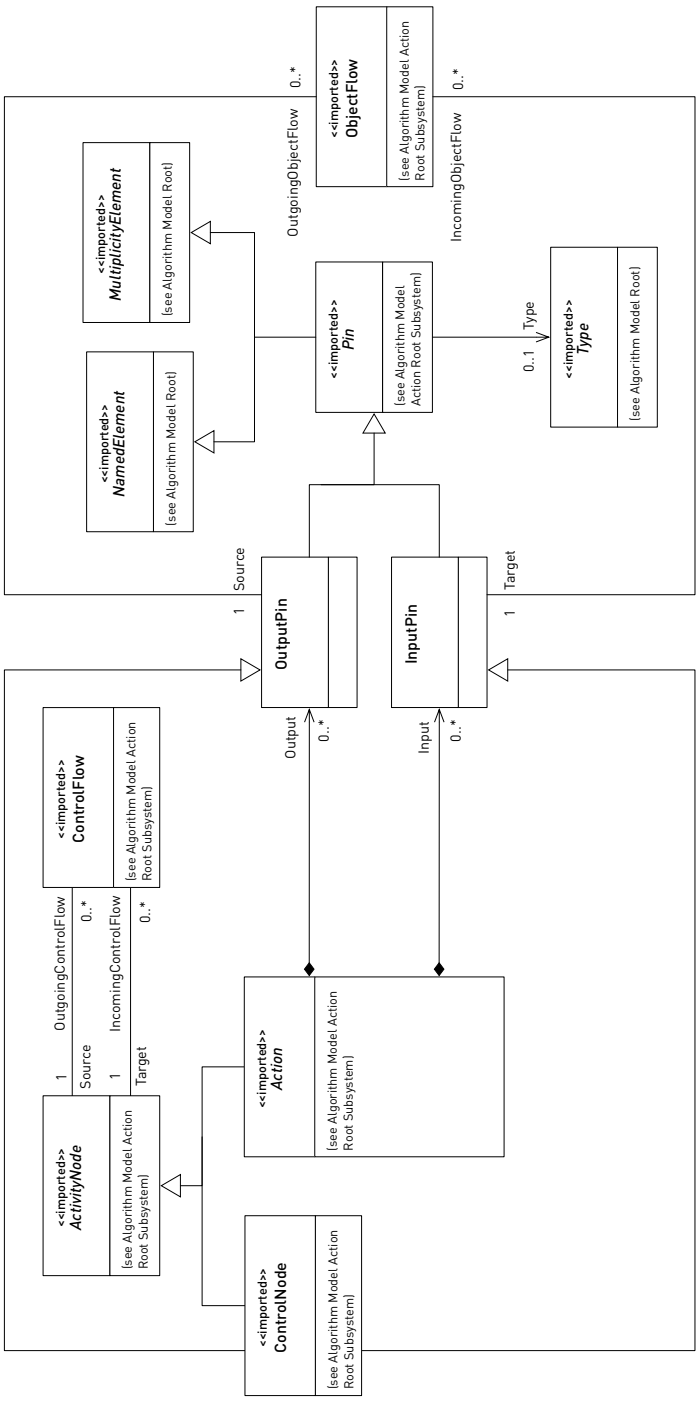


Figure B.8: Core Language Model: Action Pin Flow Subsystem.

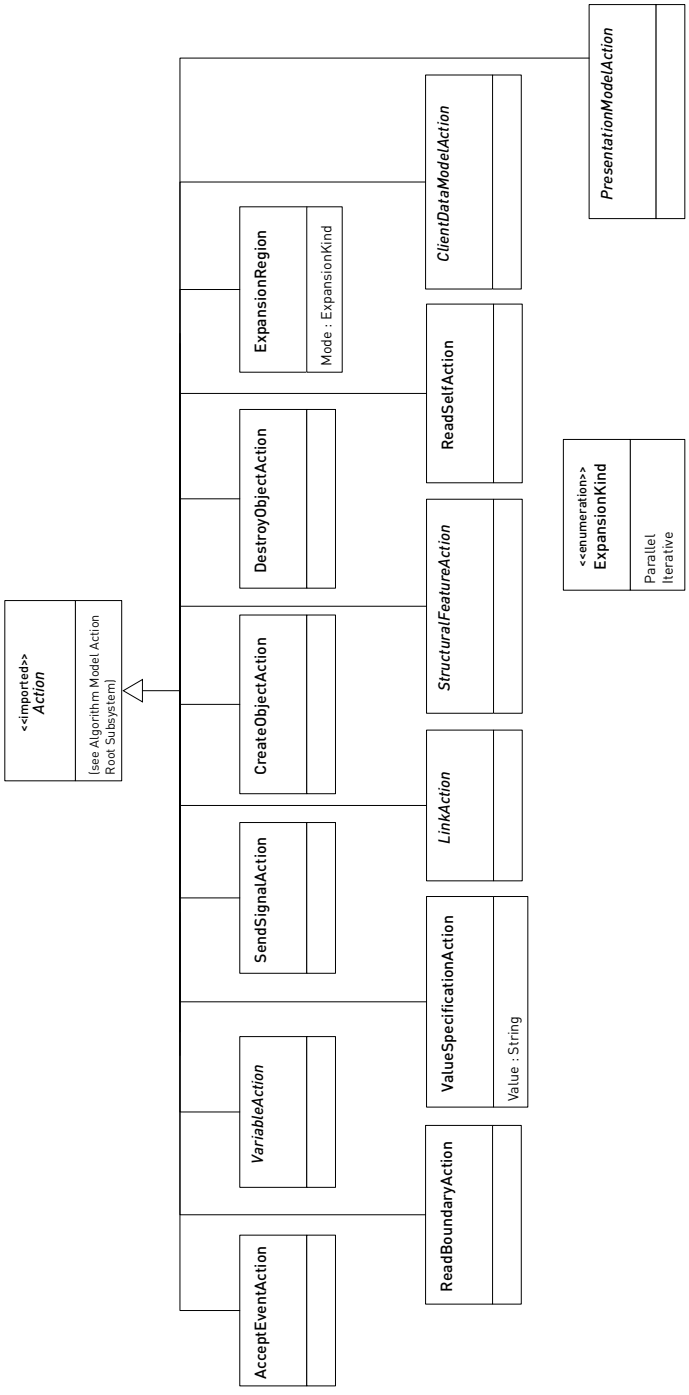


Figure B.9: Core Language Model: Action Collection.

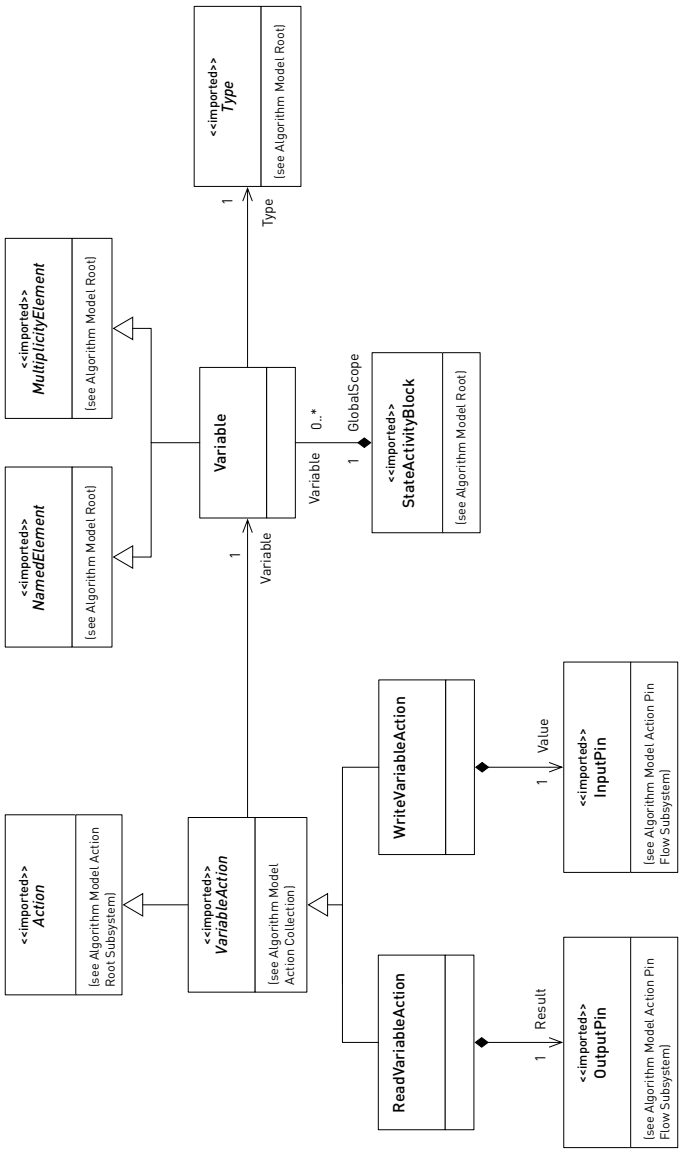


Figure B.10: Core Language Model: Variable Action Subsystem.

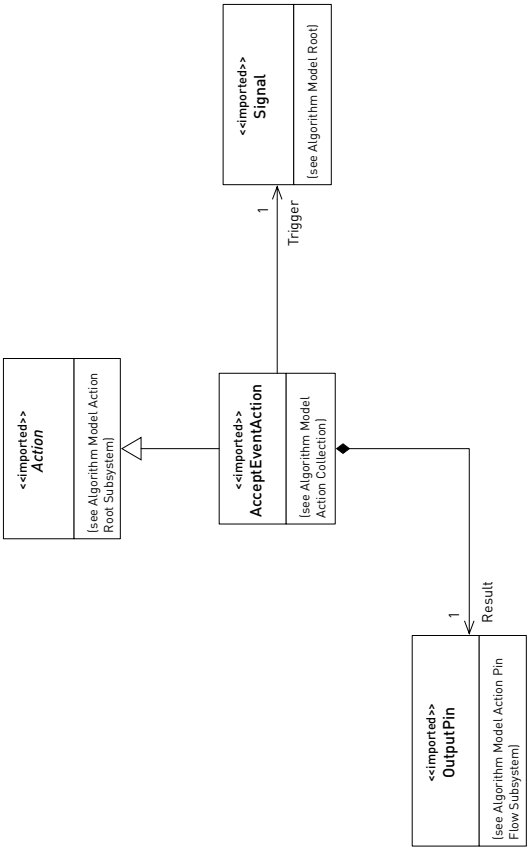


Figure B.11: Core Language Model: Accept Event Action Subsystem.

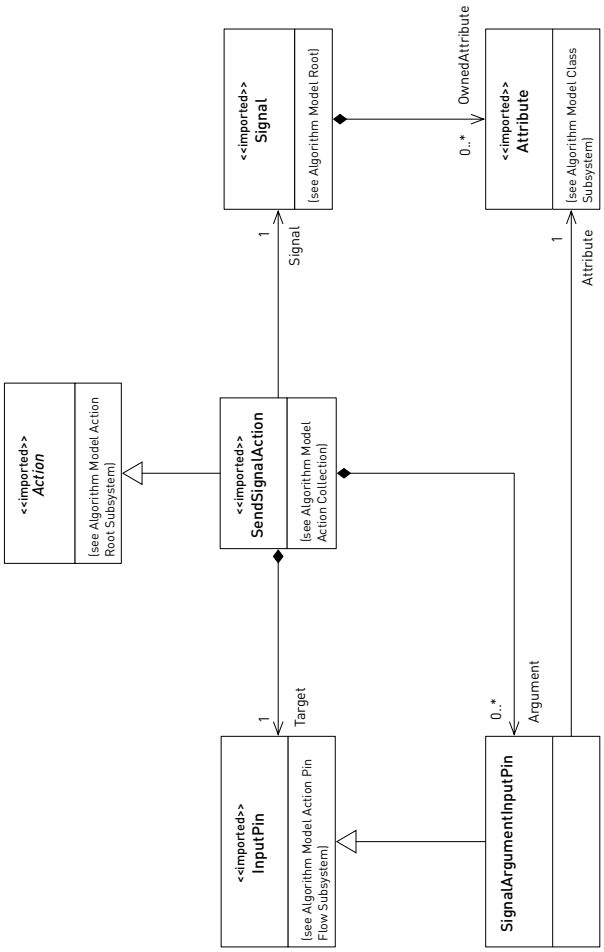


Figure B.12: Core Language Model: Send Signal Action Subsystem.

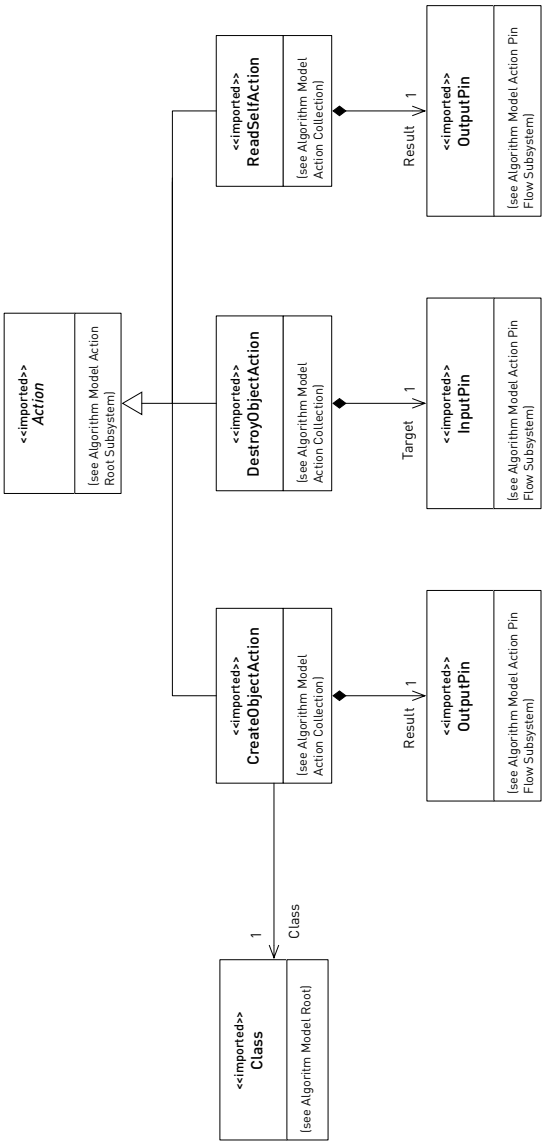


Figure B.13: Core Language Model: Object Action Subsystem.

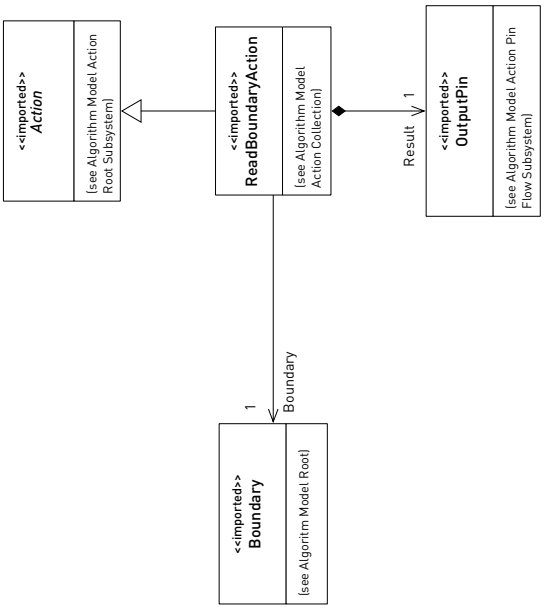


Figure B.14: Core Language Model: Read Boundary Action Subsystem.

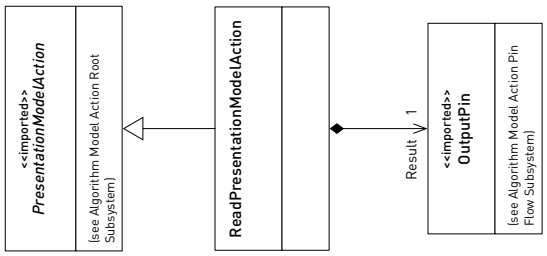


Figure B.15: Core Language Model: Read Presentation Model Action Subsystem.

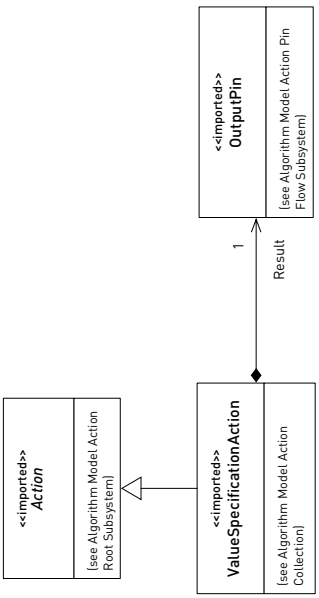


Figure B.16: Core Language Model: Value Specification Action Subsystem.

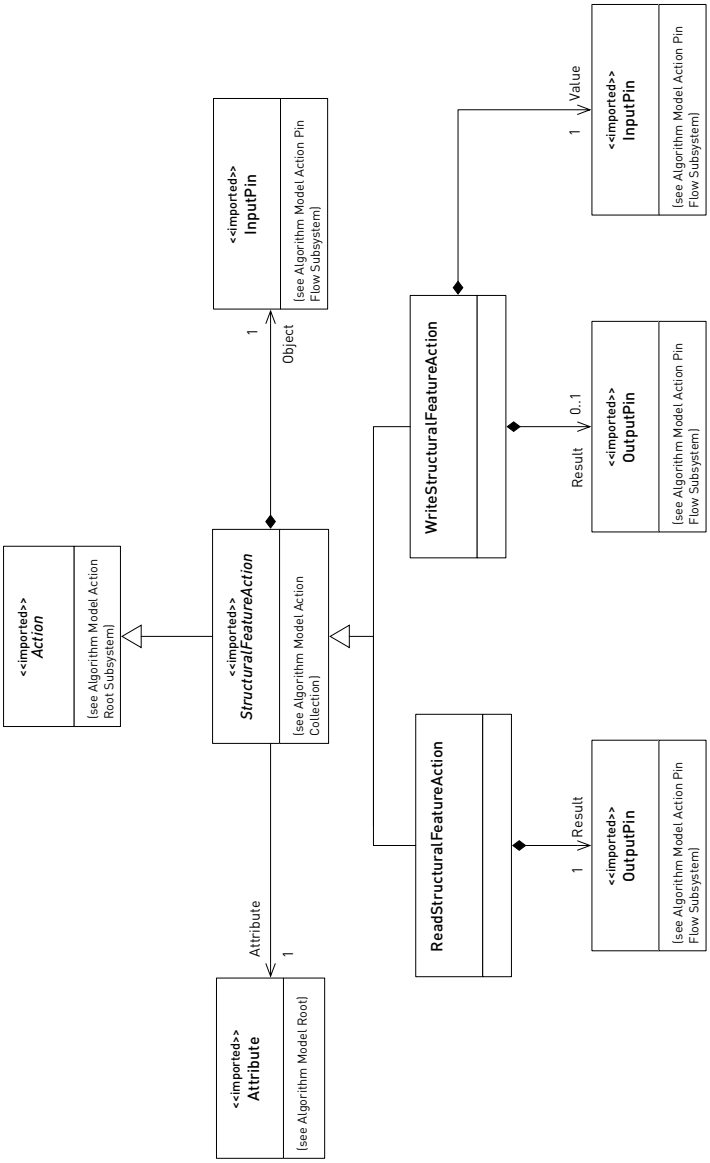


Figure B.17: Core Language Model: Structural Feature Action Subsystem.

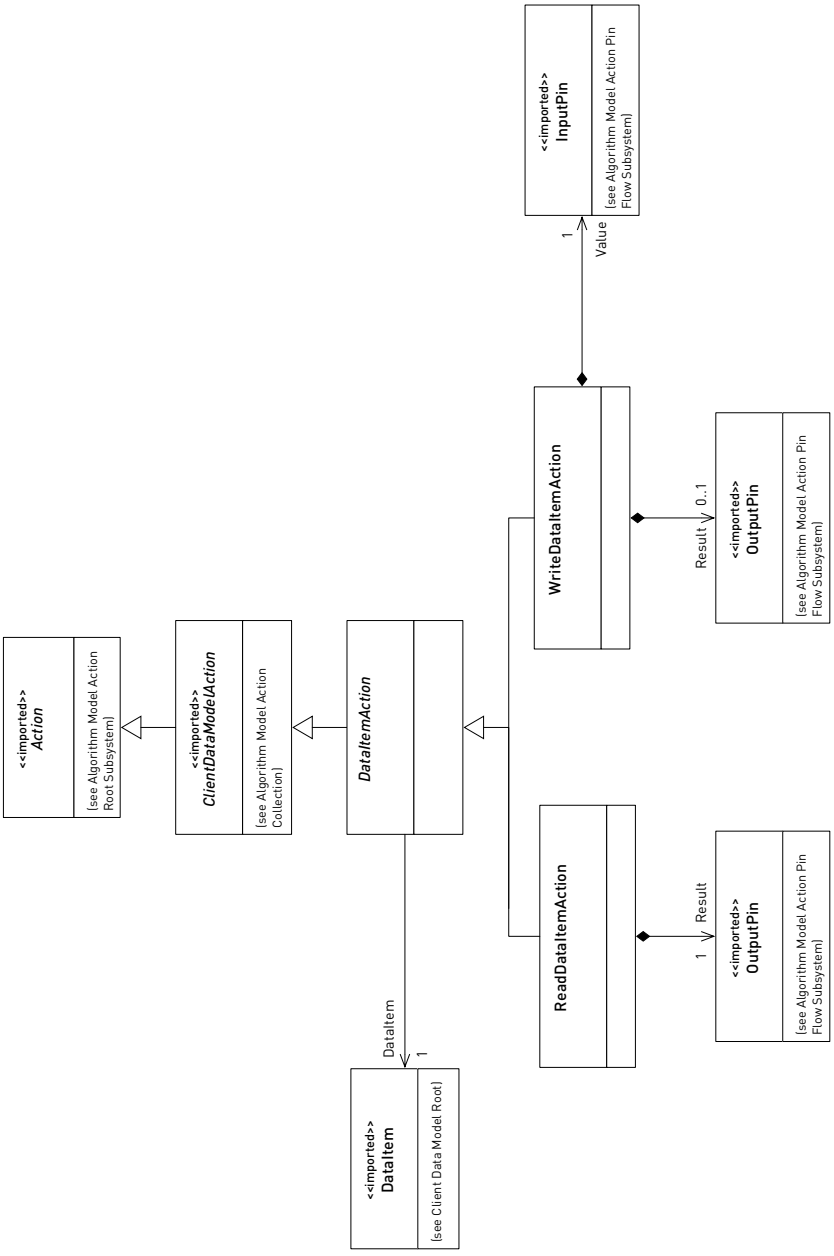


Figure B.18: Core Language Model: Data Item Action Subsystem.

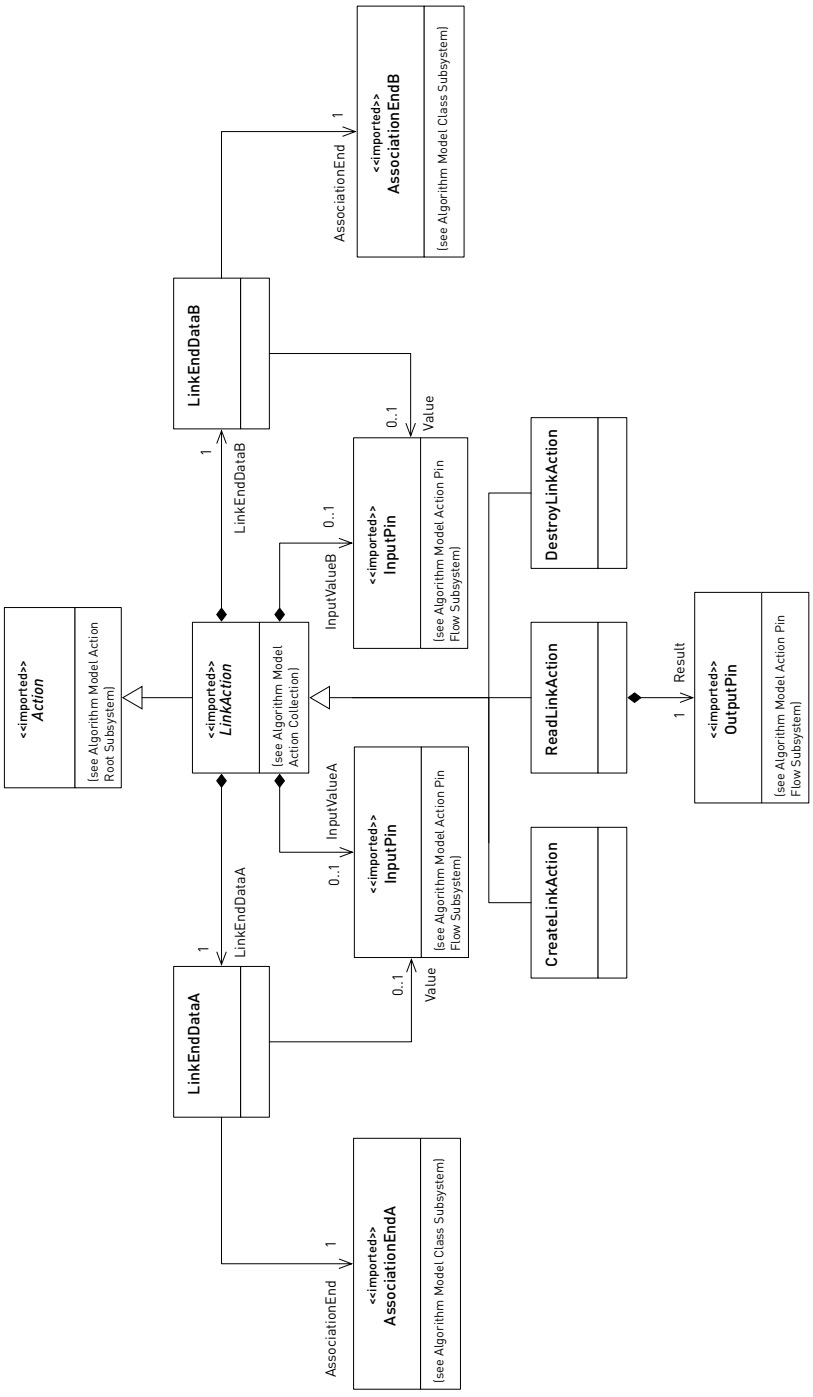


Figure B.19: Core Language Model: Link Action Subsystem.

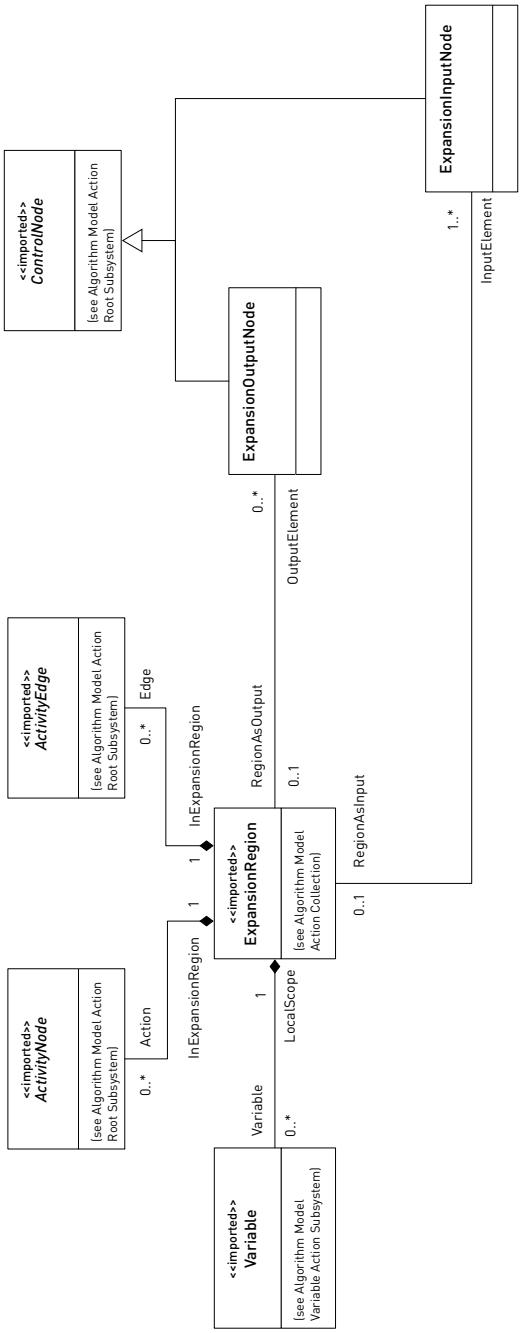


Figure B.20: Core Language Model: Expansion Region Action Subsystem.

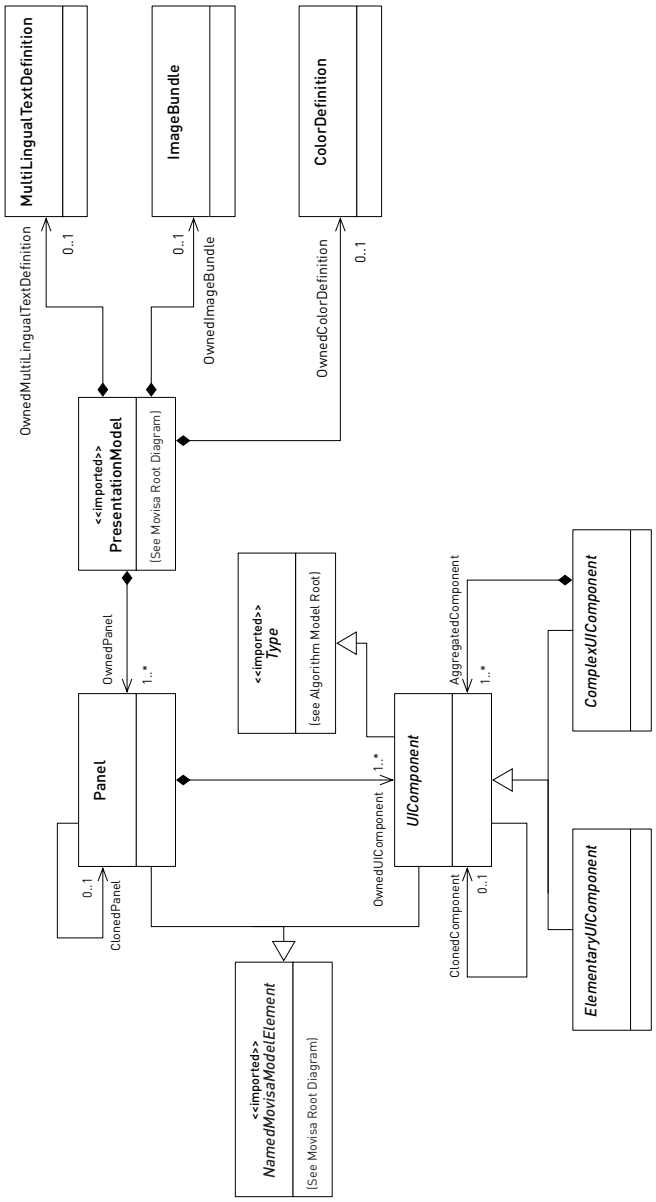


Figure B.21: Core Language Model: Presentation Model Root.

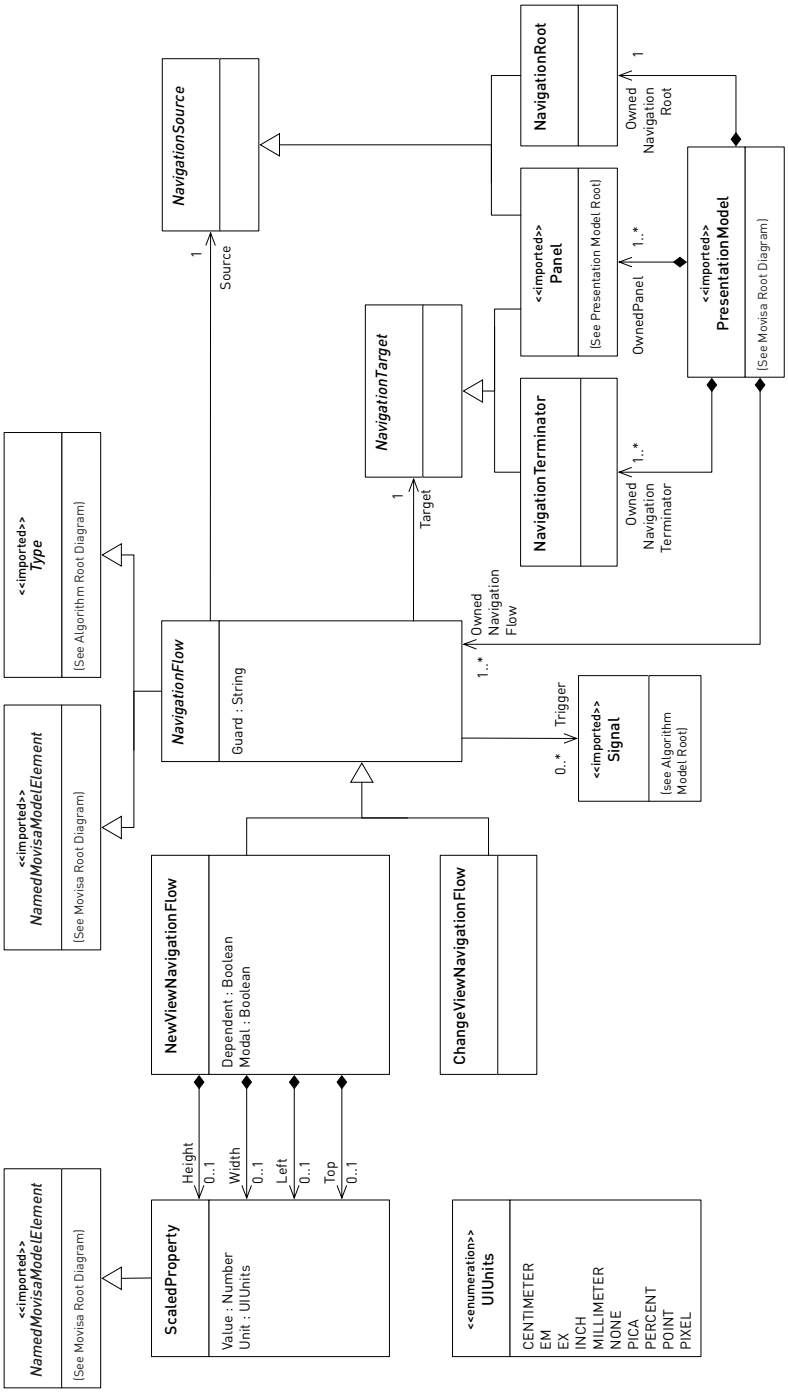


Figure B.22: Core Language Model: Navigation Subsystem.

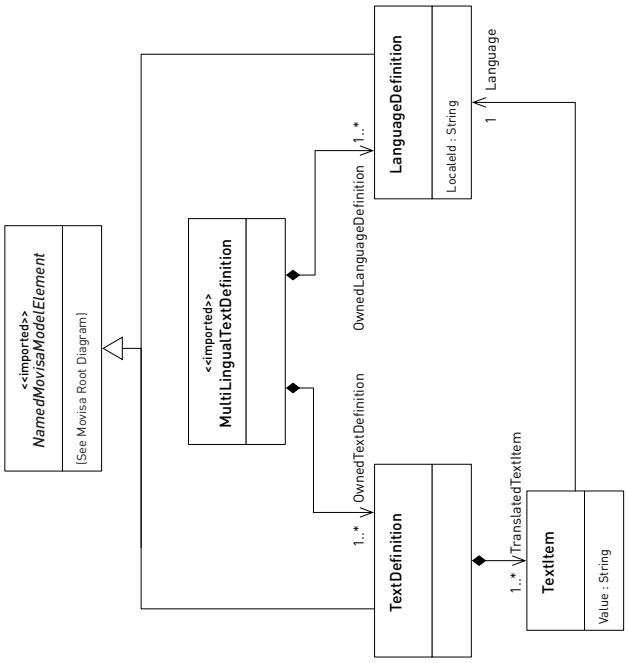


Figure B.23: Core Language Model: Multi Lingual Text Definition Subsystem.

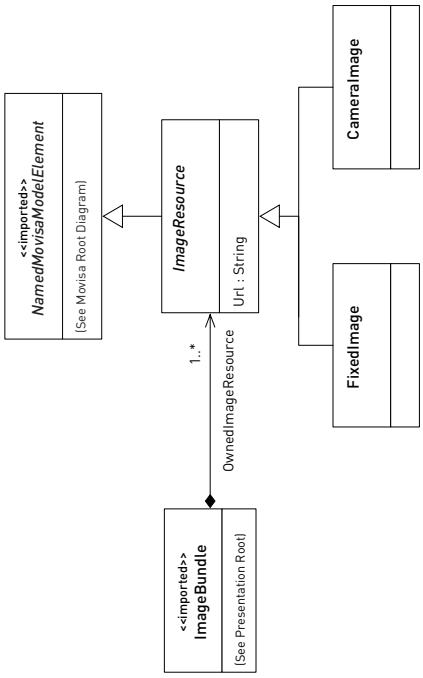


Figure B.24: Core Language Model: Image Bundle Subsystem.

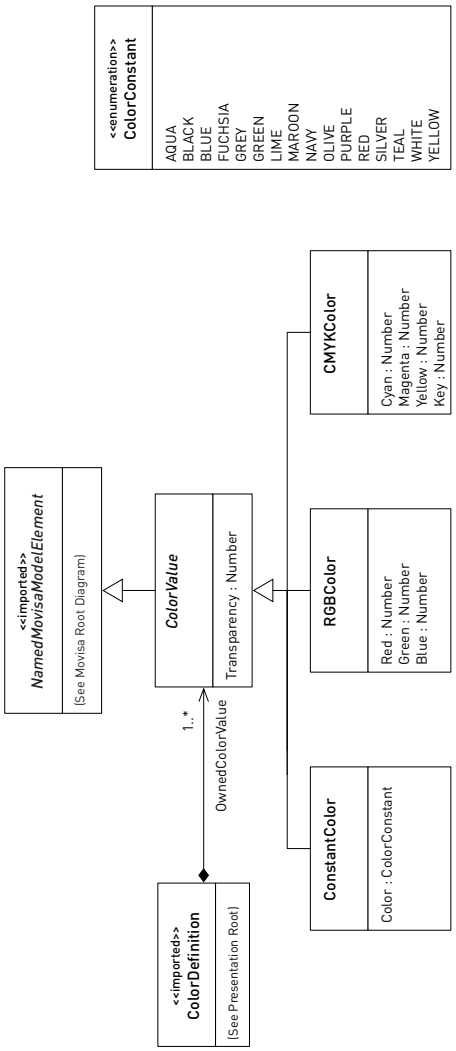


Figure B.25: Core Language Model: Color Definition Subsystem.

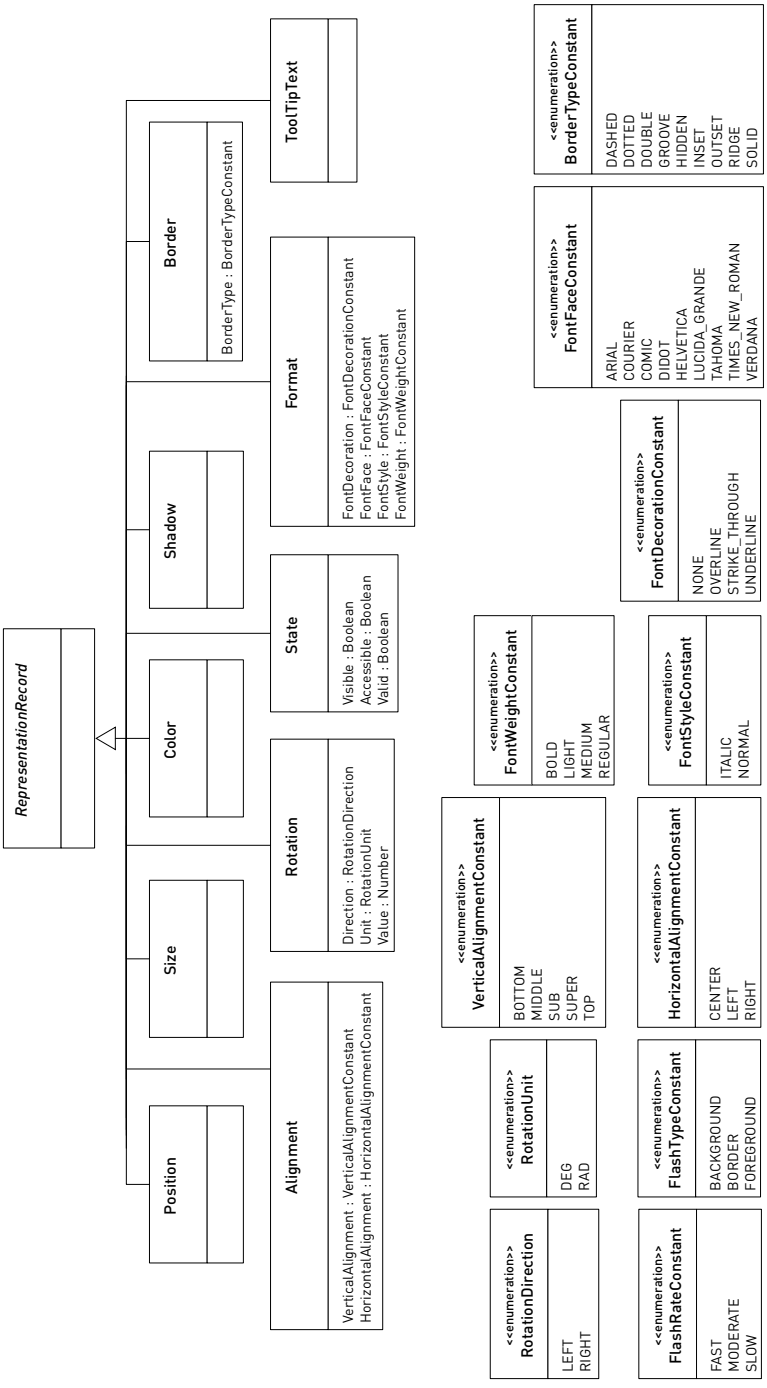


Figure B.26: Core Language Model: Representation Record Subsystem.

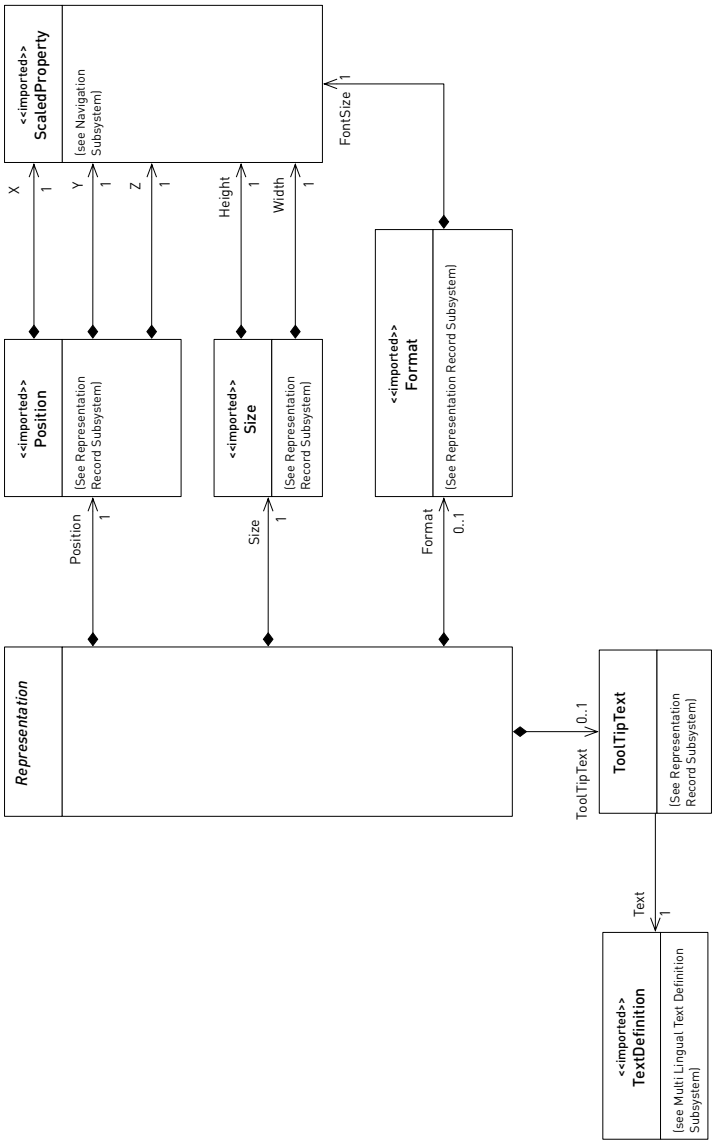


Figure B.27: Core Language Model: Representation Subsystem (1).

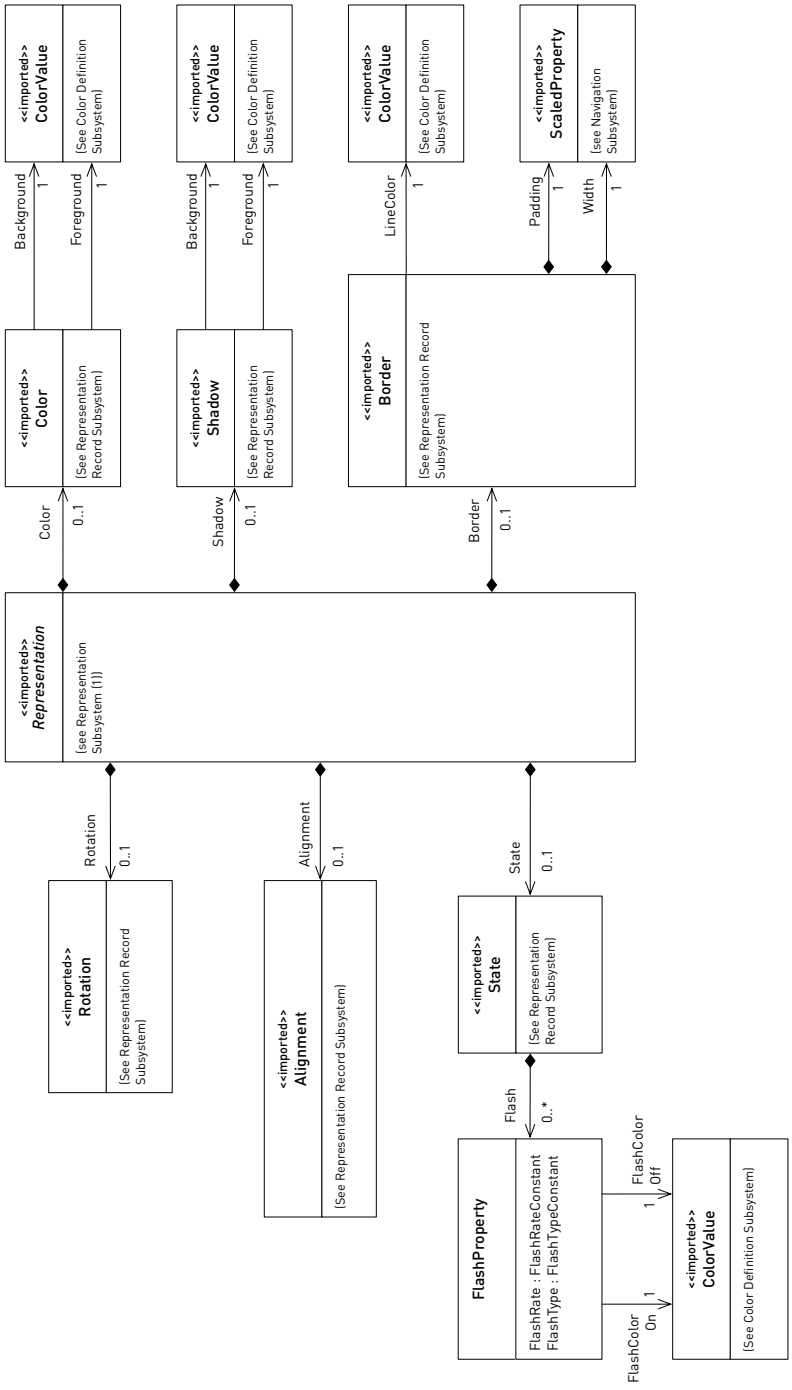


Figure B.28: Core Language Model: Representation Subsystem (2).

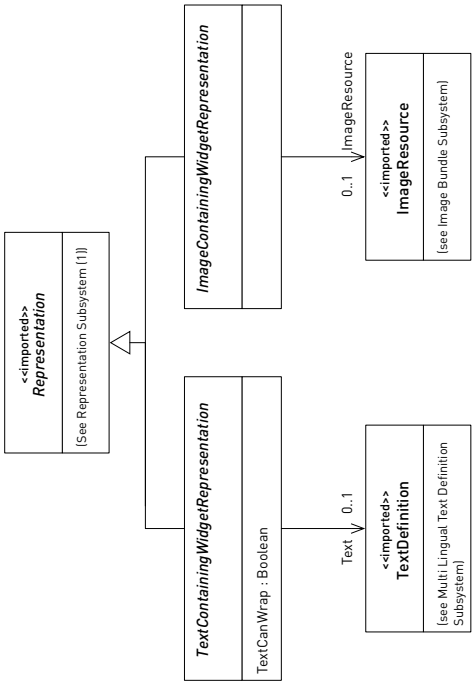


Figure B.29: Core Language Model: Representation Concrete Type Subsystem.

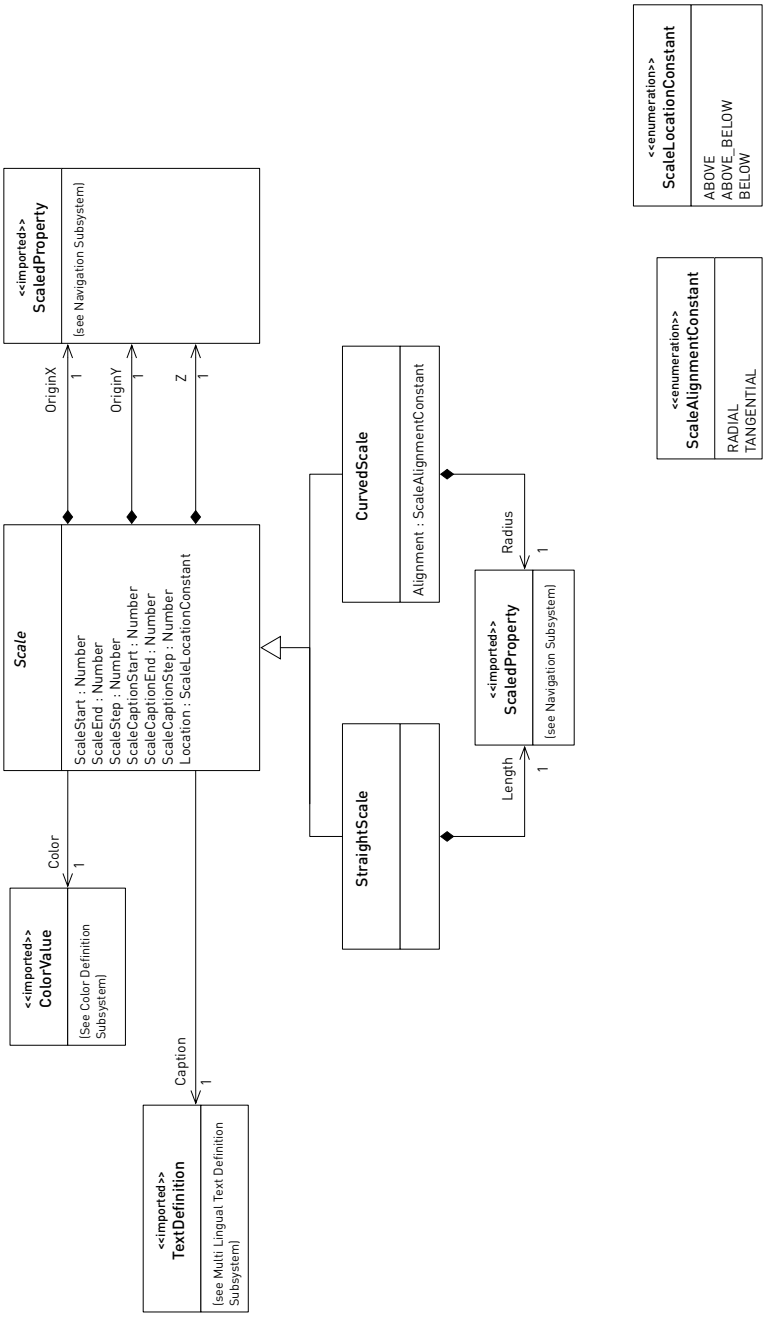


Figure B.30: Core Language Model: Representation Scale Subsystem.

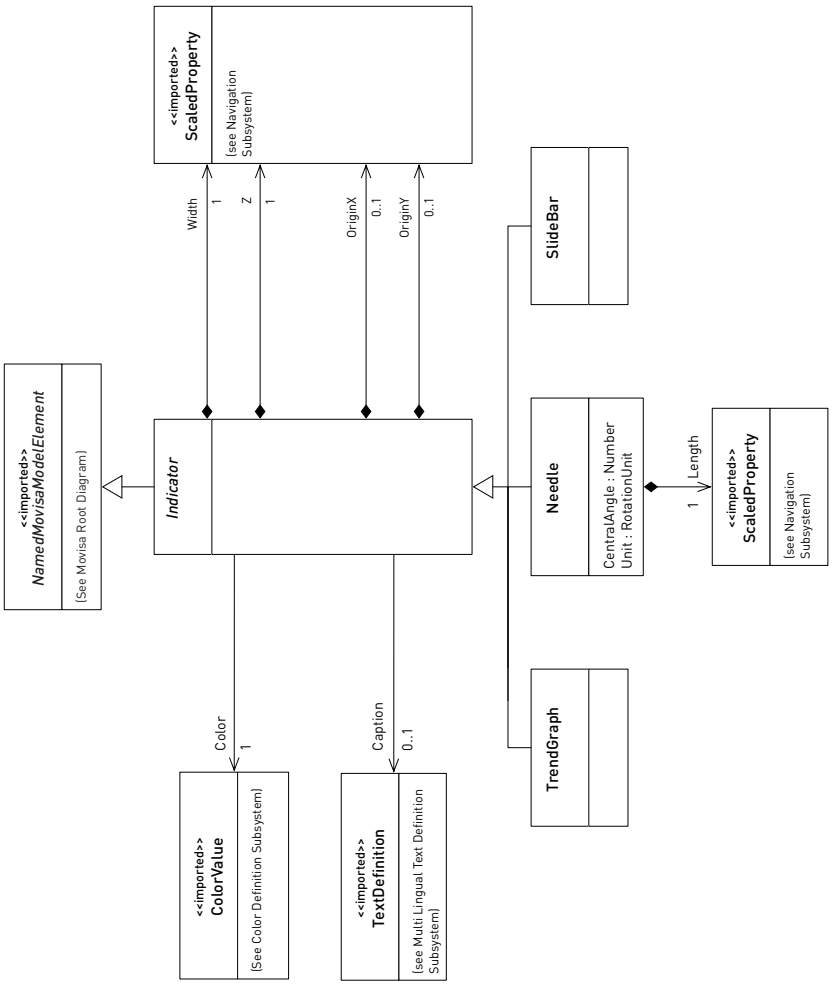


Figure B.31: Core Language Model: Representation Indicator Subsystem.

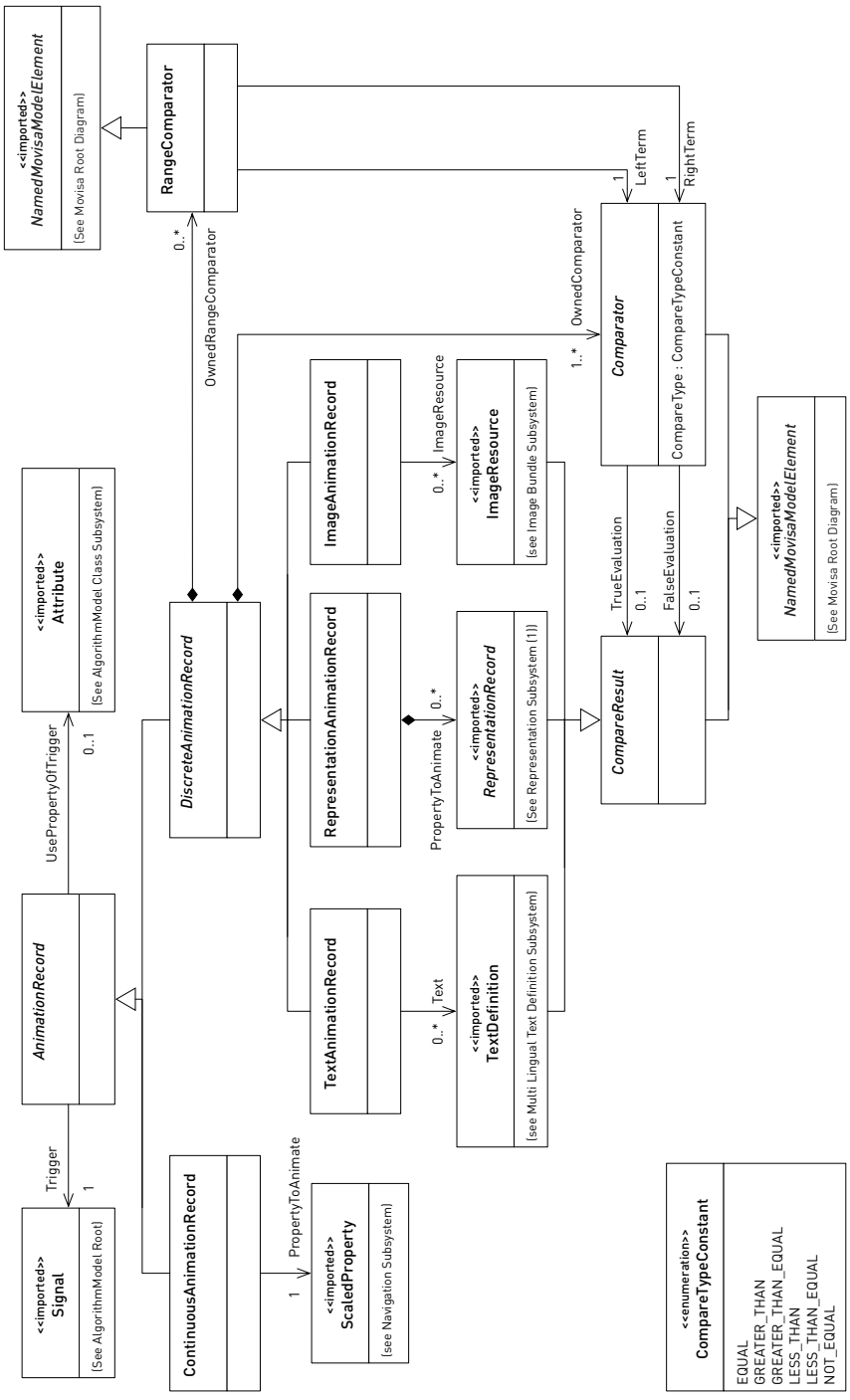


Figure B.32: Core Language Model: Animation Subsystem (1).

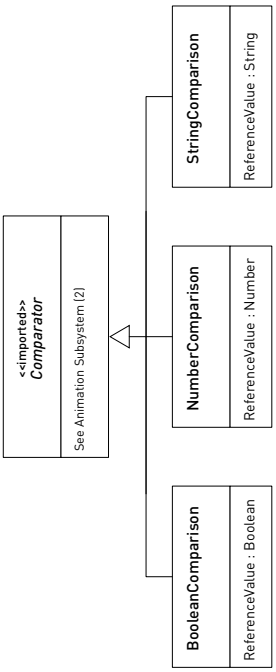


Figure B.33: Core Language Model: Animation Subsystem (2).

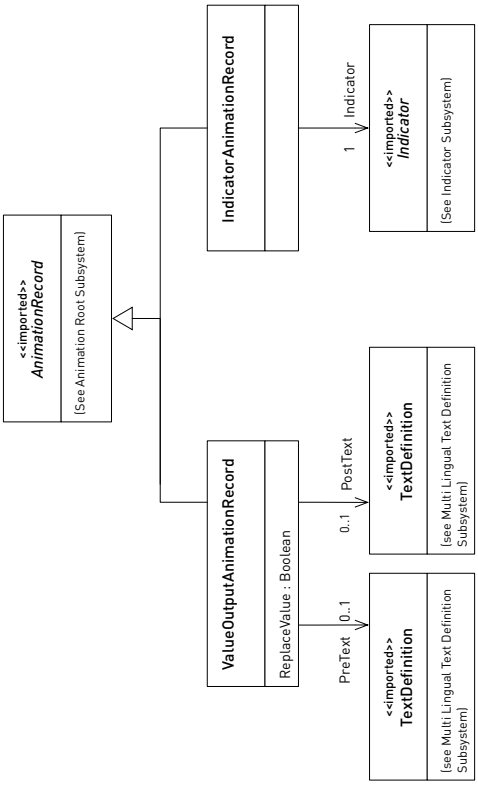


Figure B.34: Core Language Model: Animation Subsystem (3).

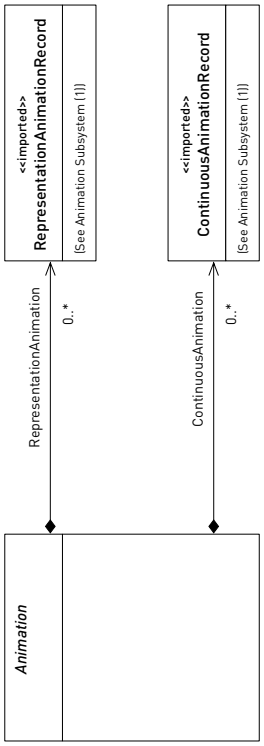


Figure B.35: Core Language Model: Animation Subsystem (4).

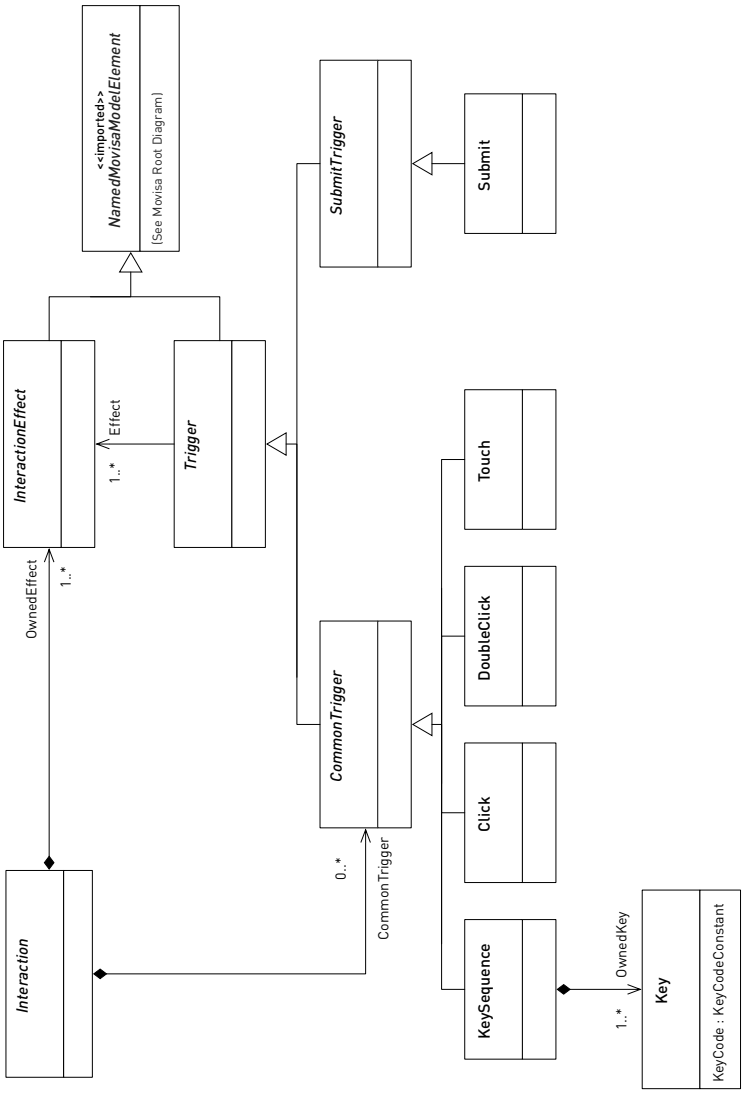


Figure B.36: Core Language Model: Interaction Subsystem.

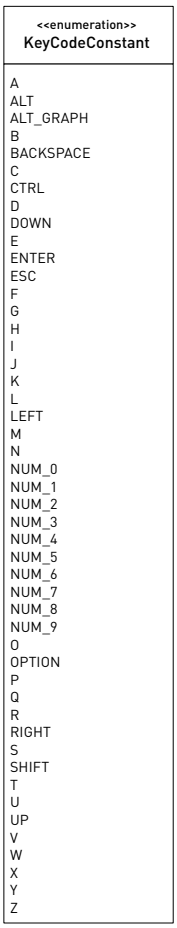


Figure B.37: Core Language Model: Interaction Subsystem (Key Code Constants).

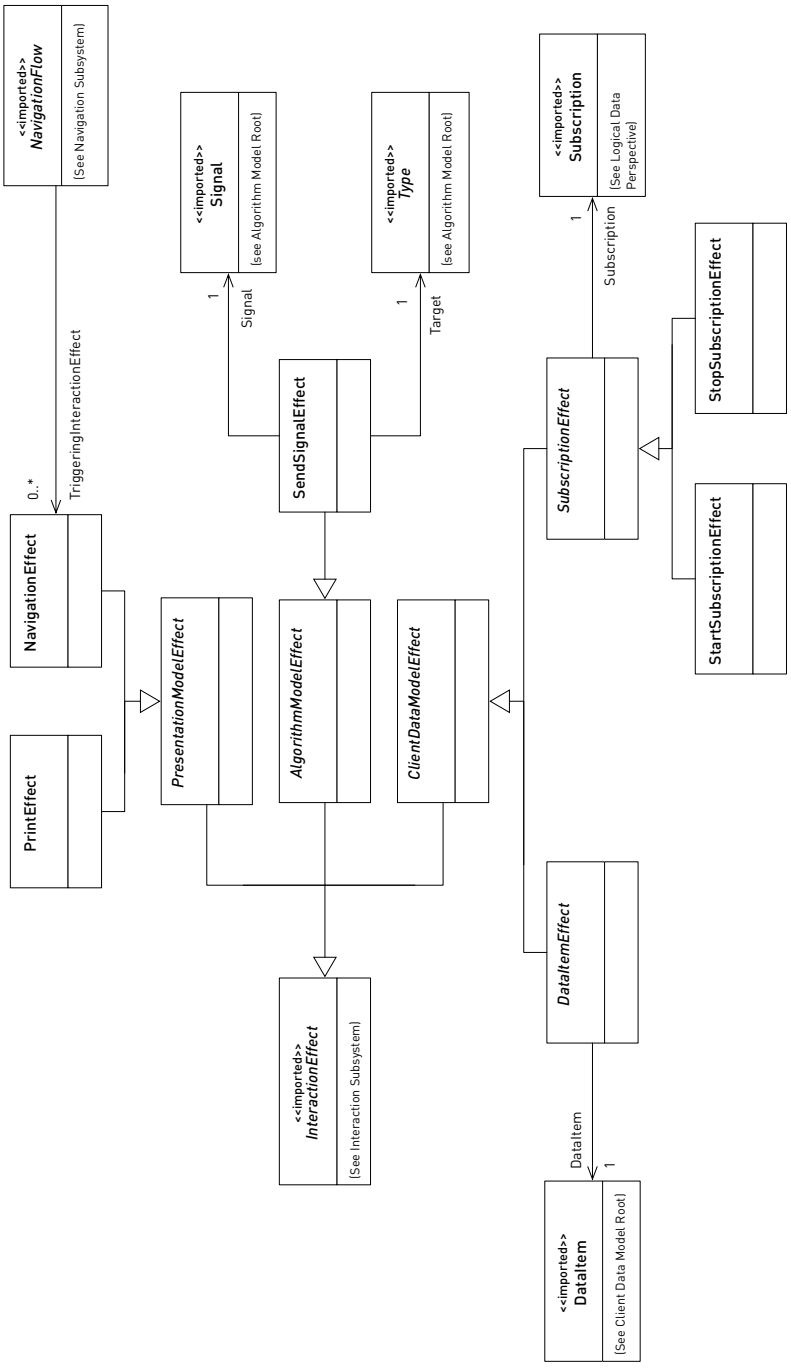


Figure B.38: Core Language Model: Interaction Effect Subsystem (1).

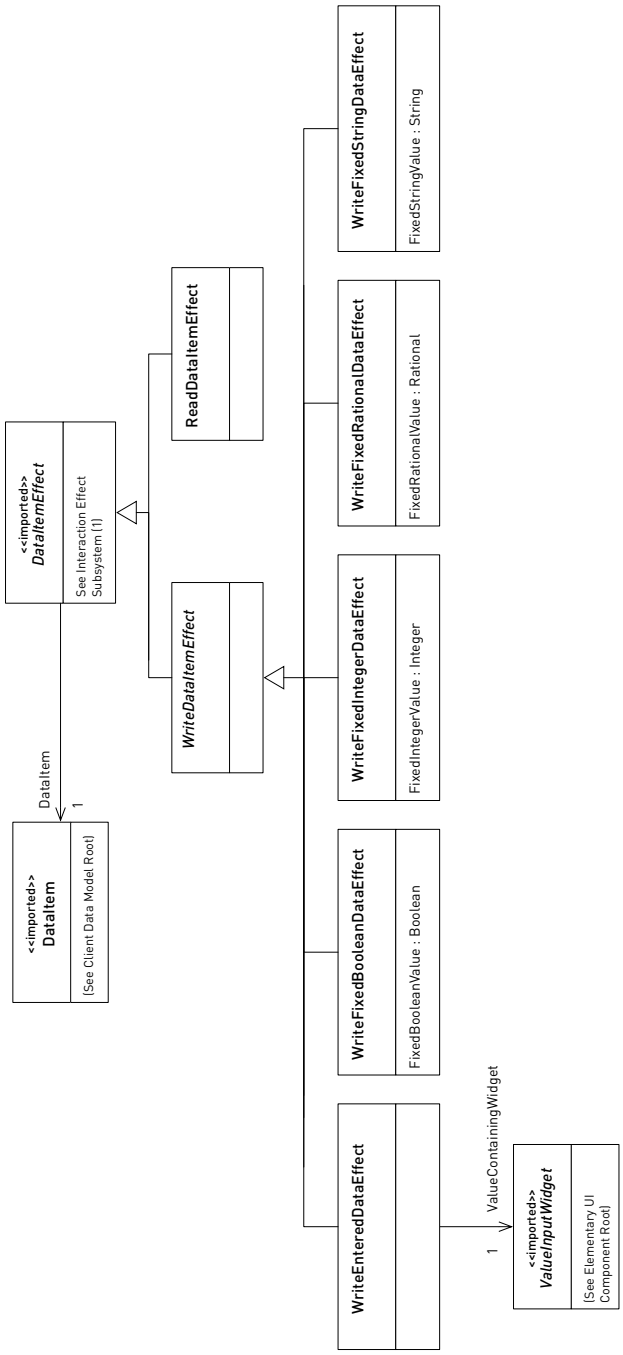


Figure B.39: Core Language Model: Interaction Effect Subsystem (2).

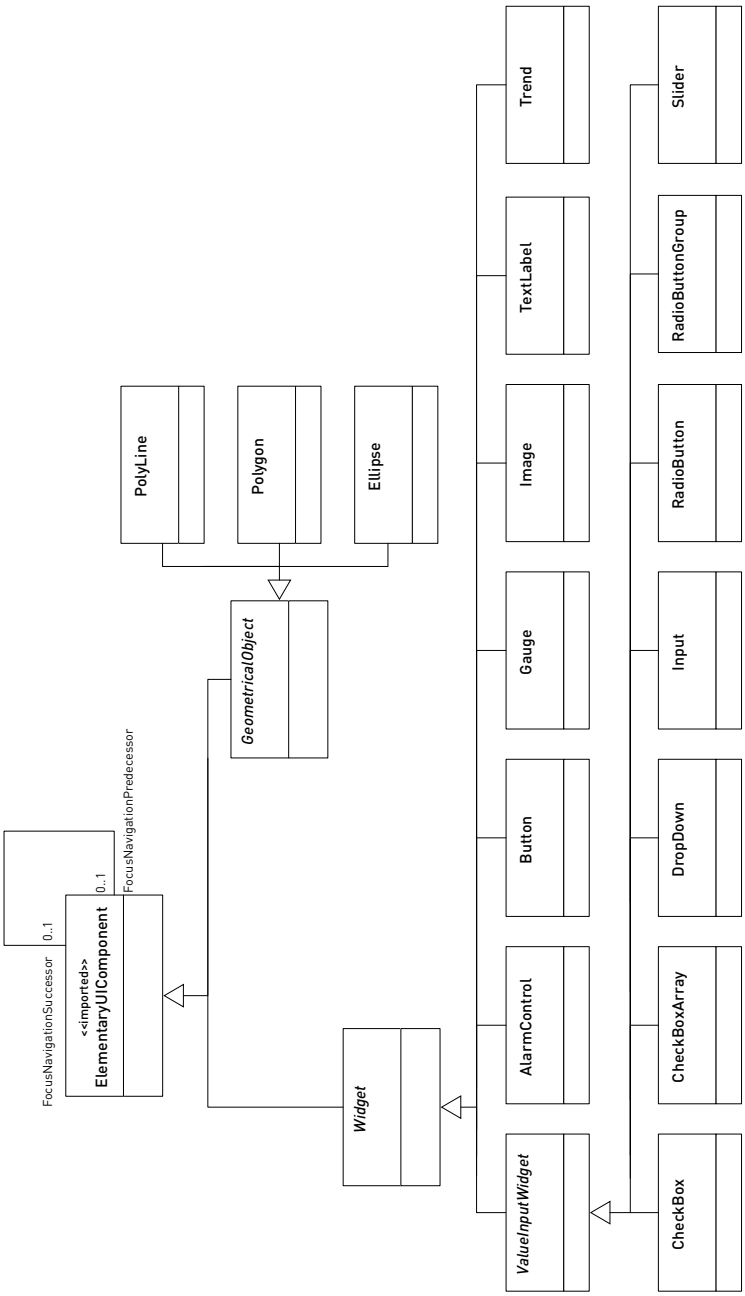


Figure B.40: Core Language Model: Elementary UI Component Root.

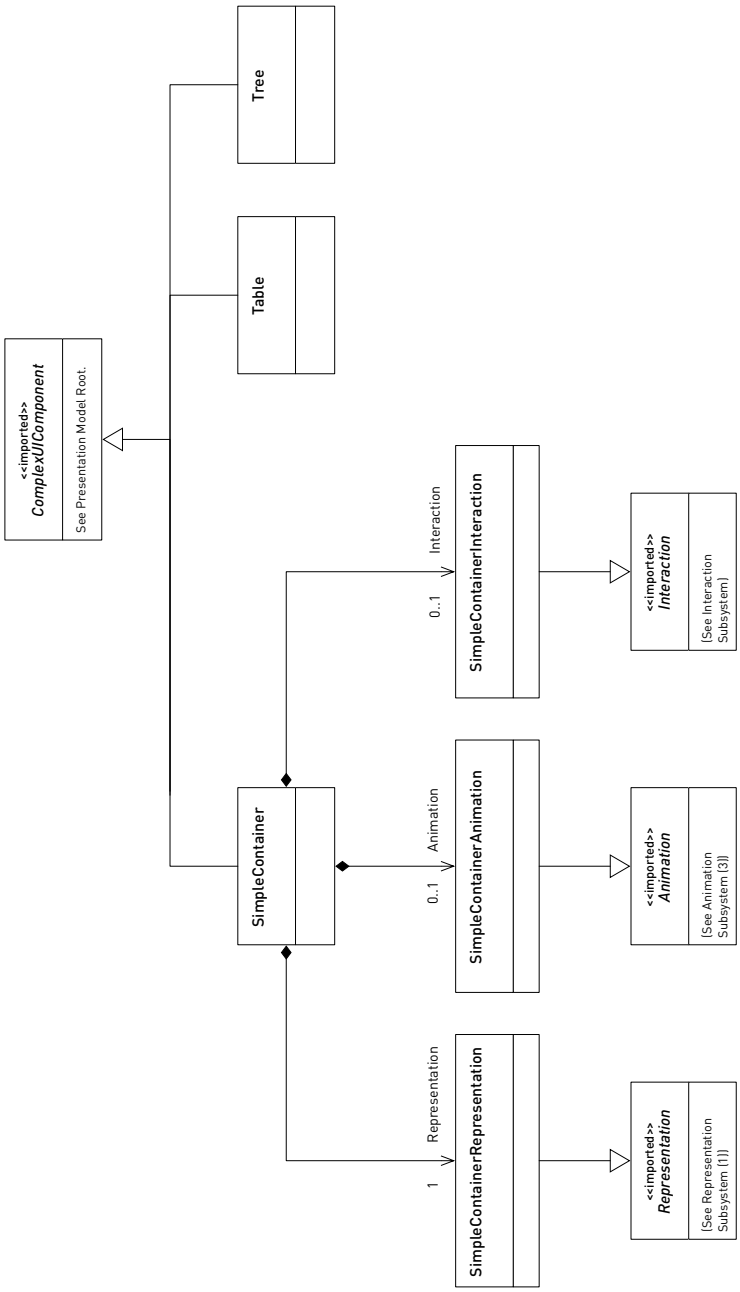


Figure B.41: Core Language Model: Complex UI Component Root.

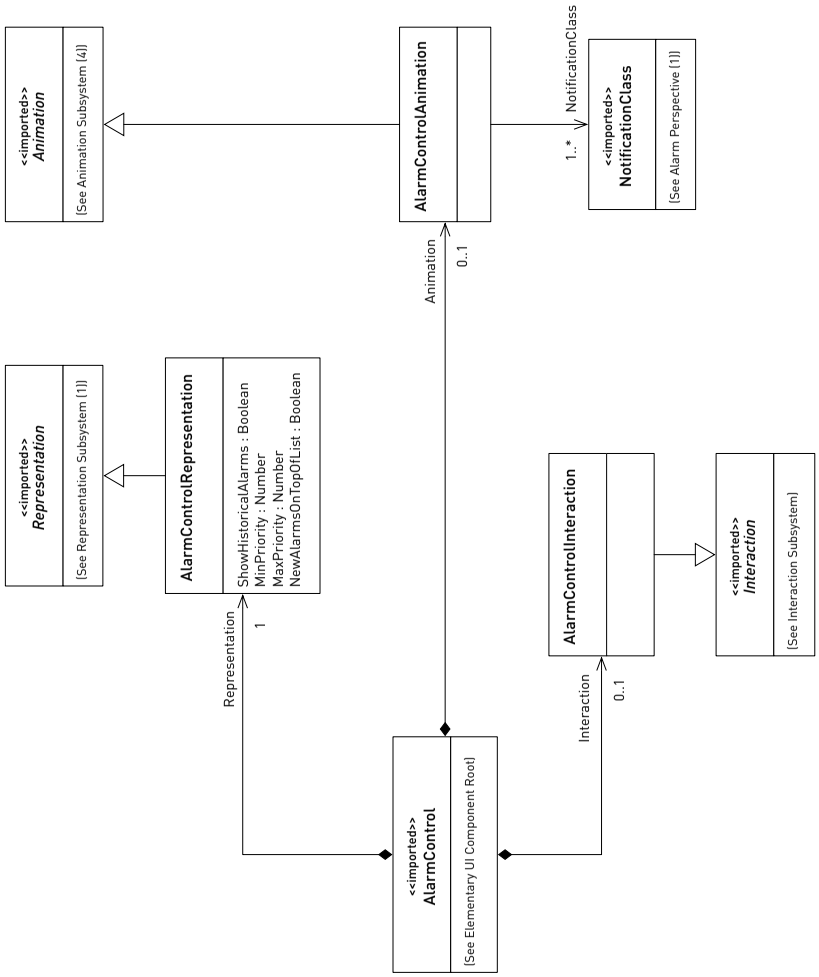


Figure B.42: Core Language Model: Alarm Control Widget.

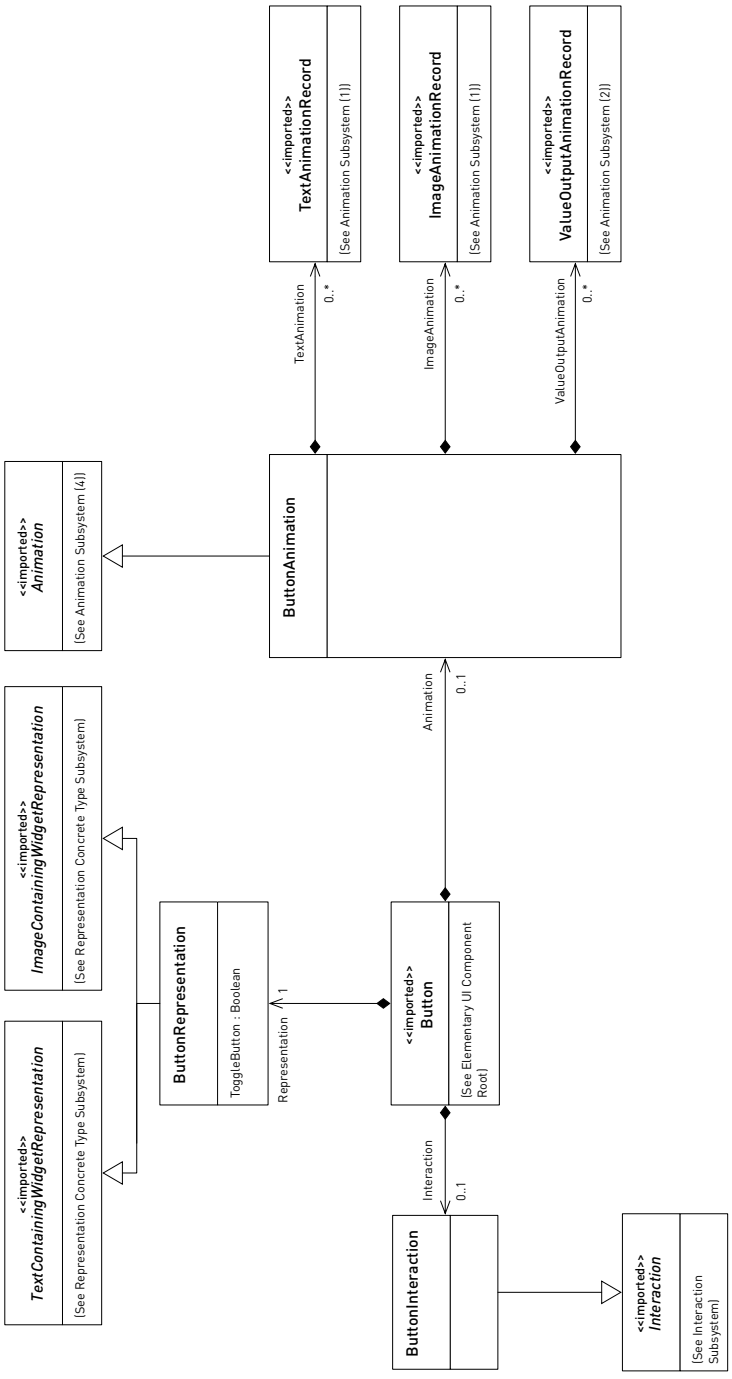


Figure B.43: Core Language Model: Button Widget.

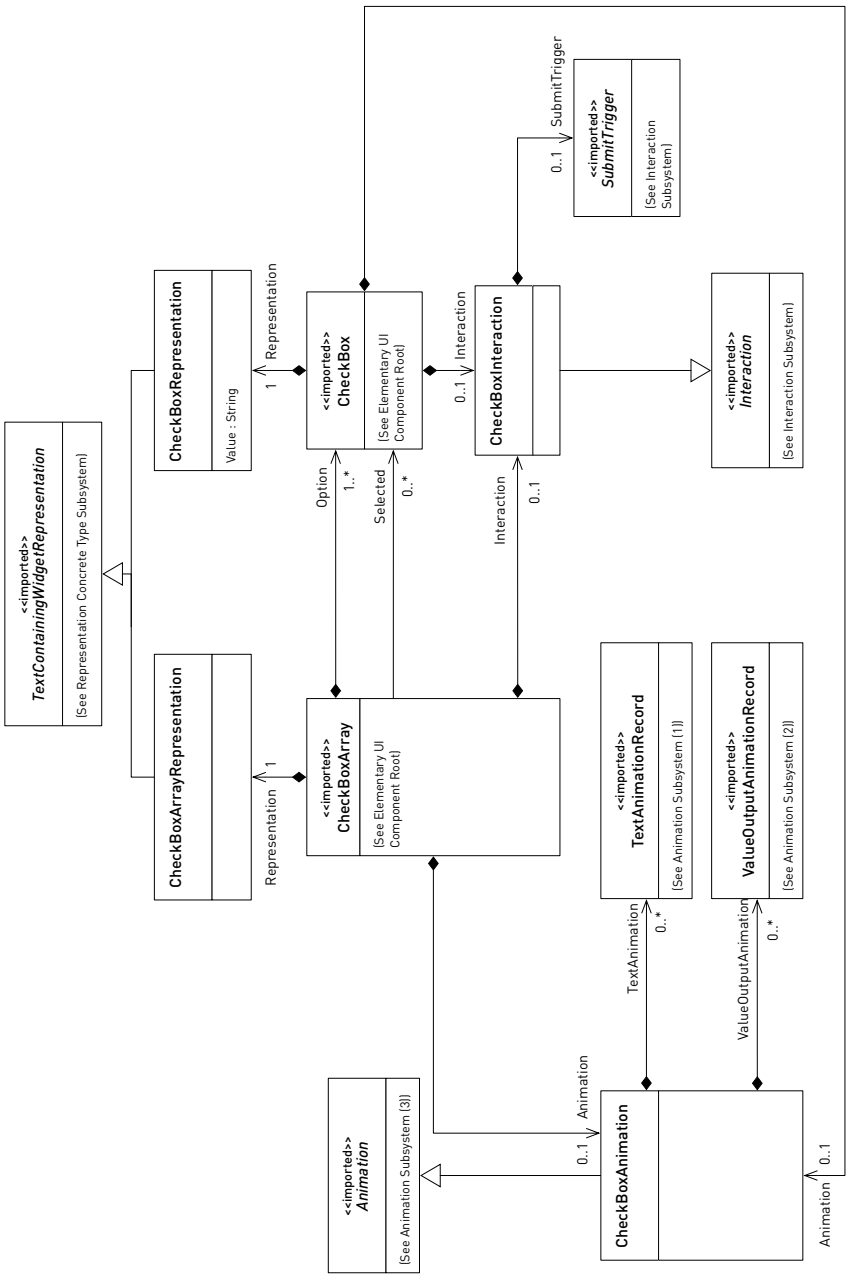


Figure B.44: Core Language Model: Check Box Widget.

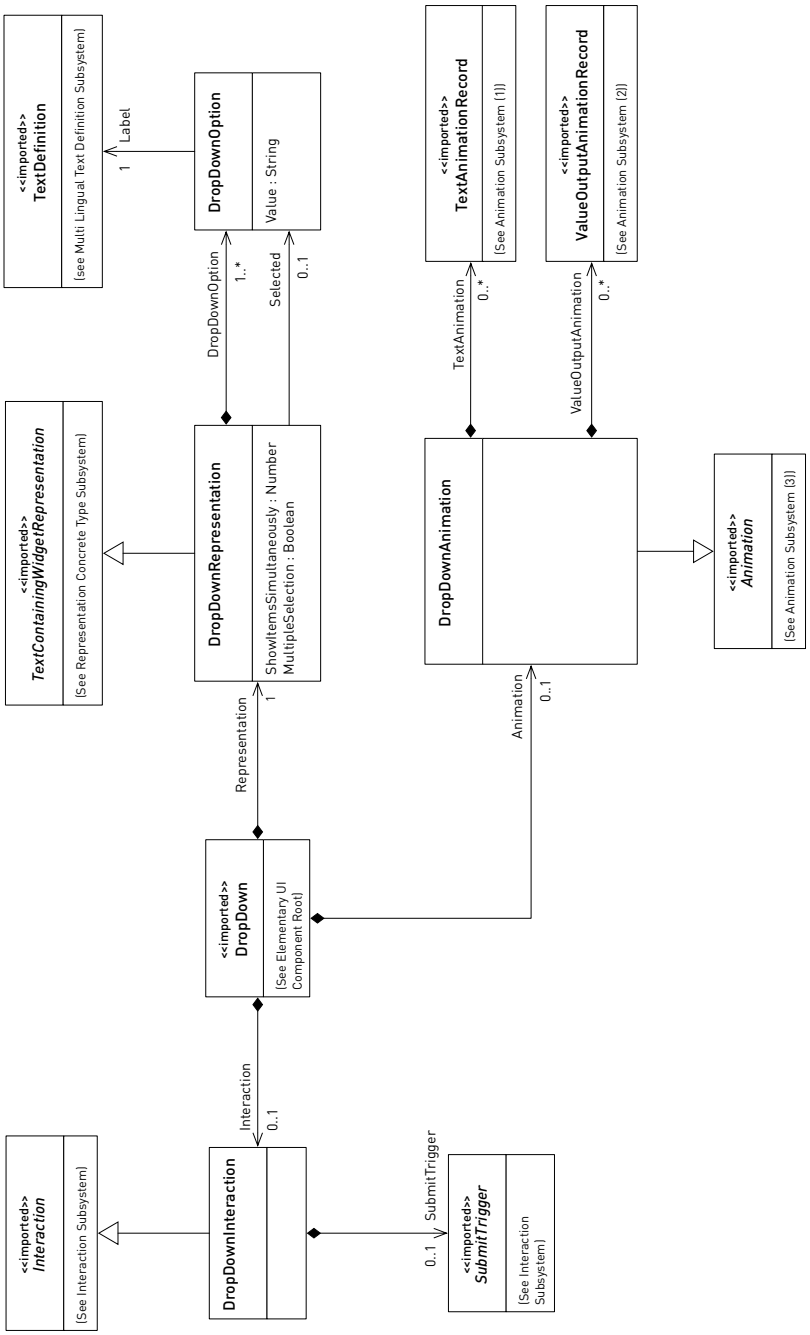


Figure B.45: Core Language Model: Drop Down Widget.

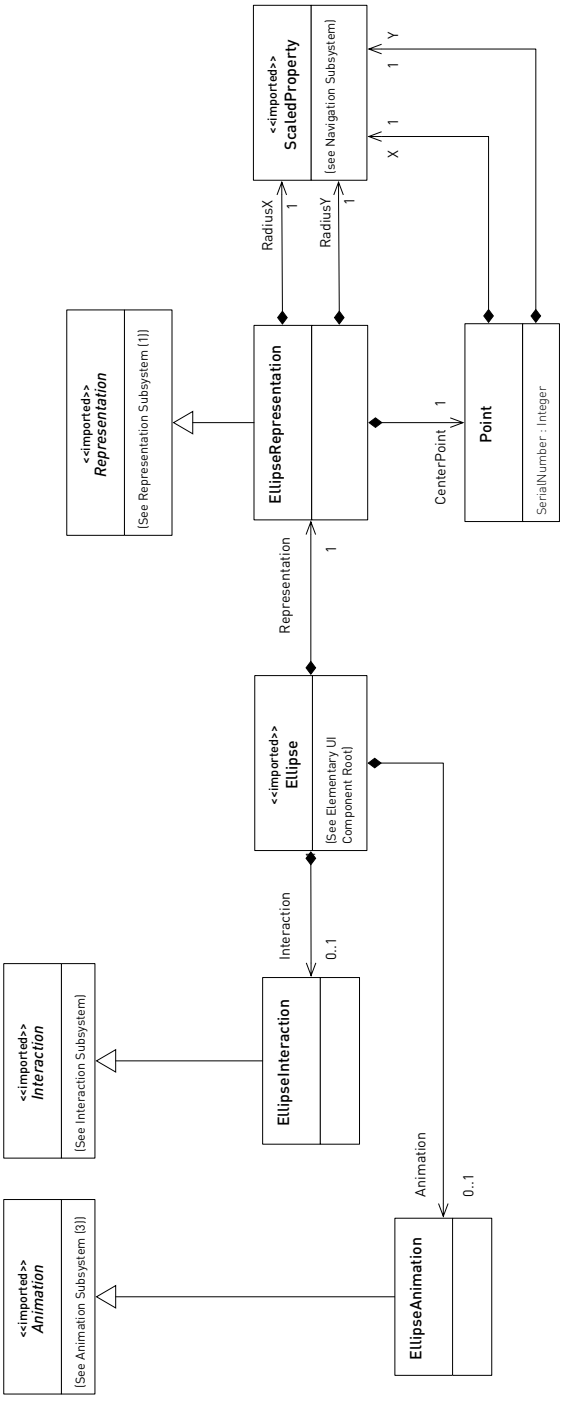


Figure B.46: Core Language Model: Ellipse Geometrical Object.

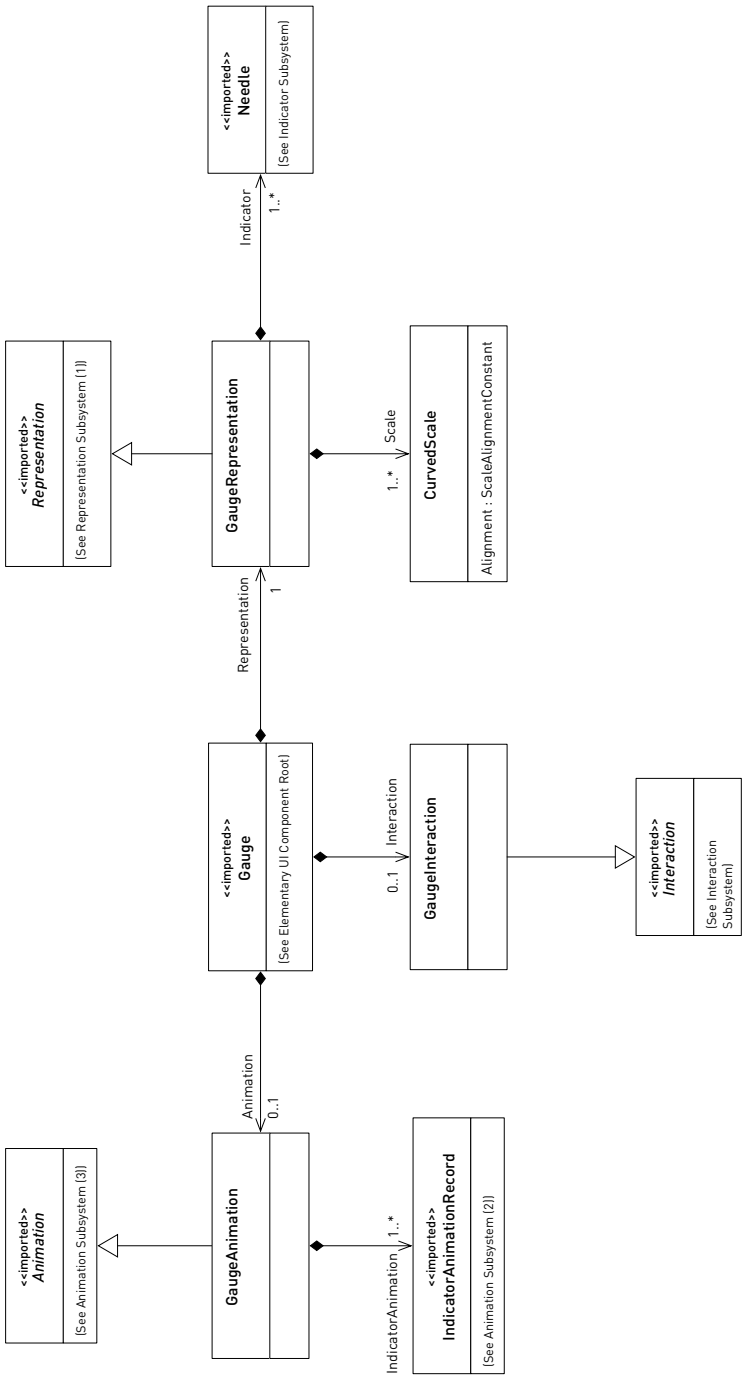


Figure B.47: Core Language Model: Gauge Widget.

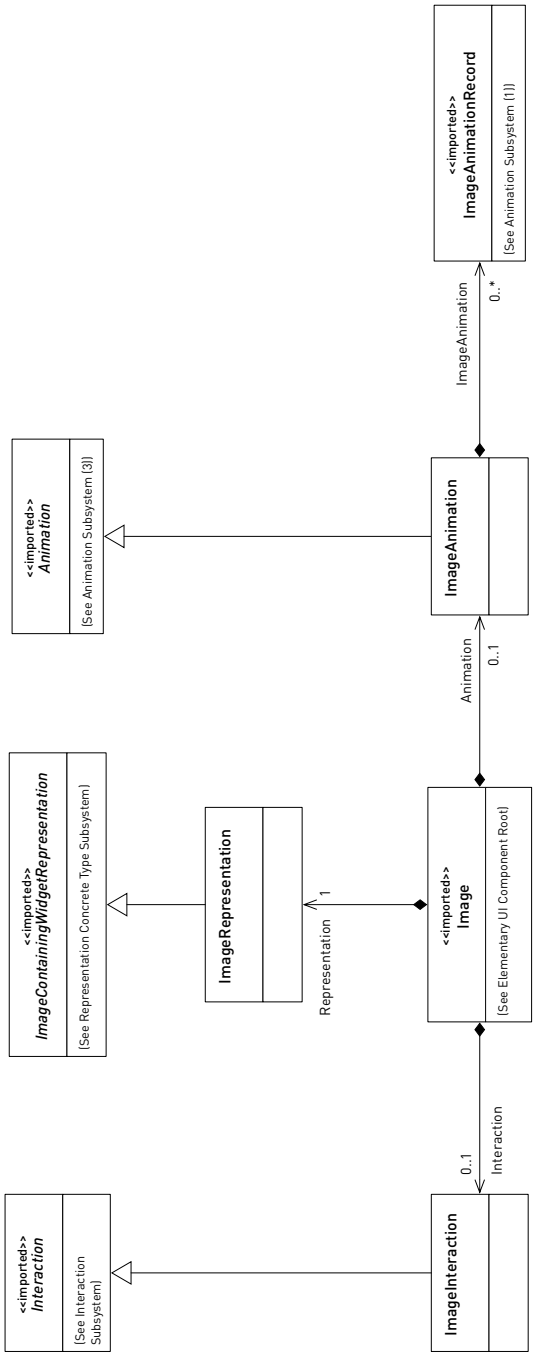


Figure B.48: Core Language Model: Image Widget.

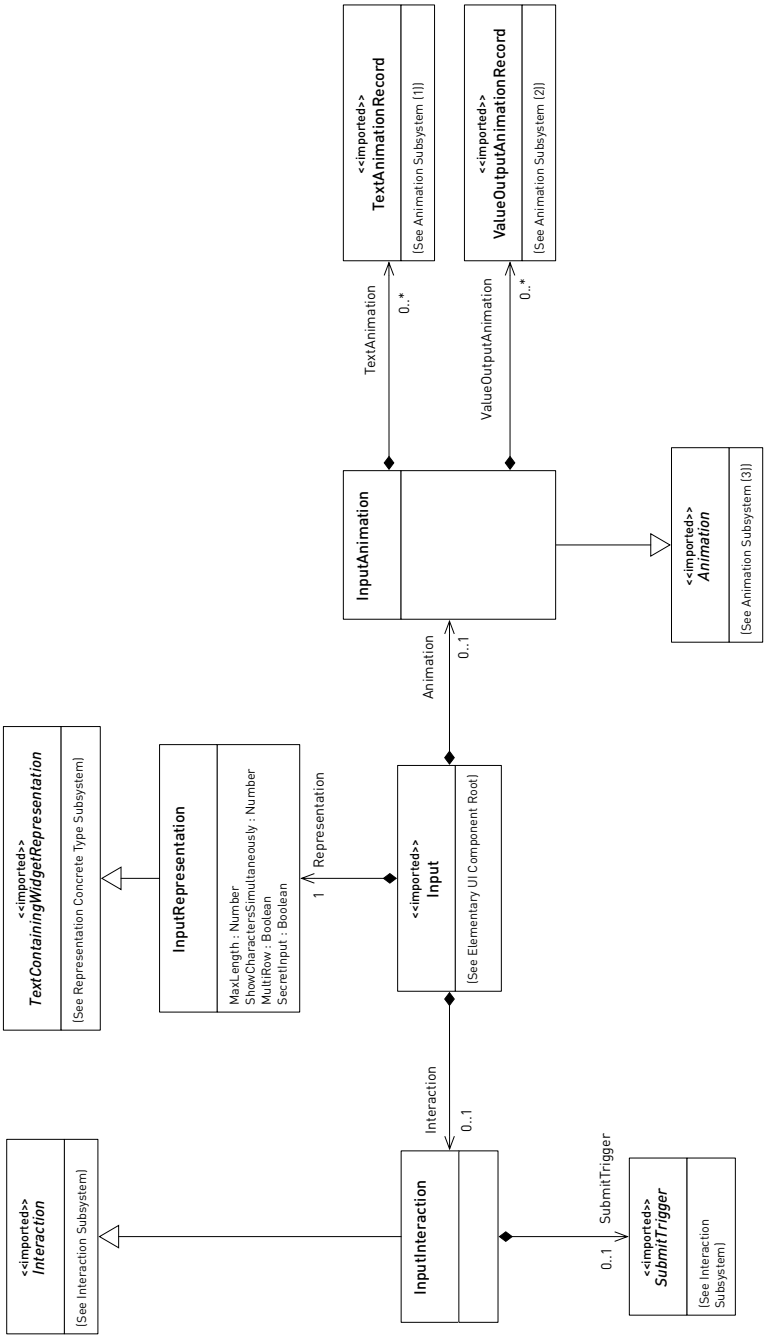


Figure B.49: Core Language Model: Input Widget.

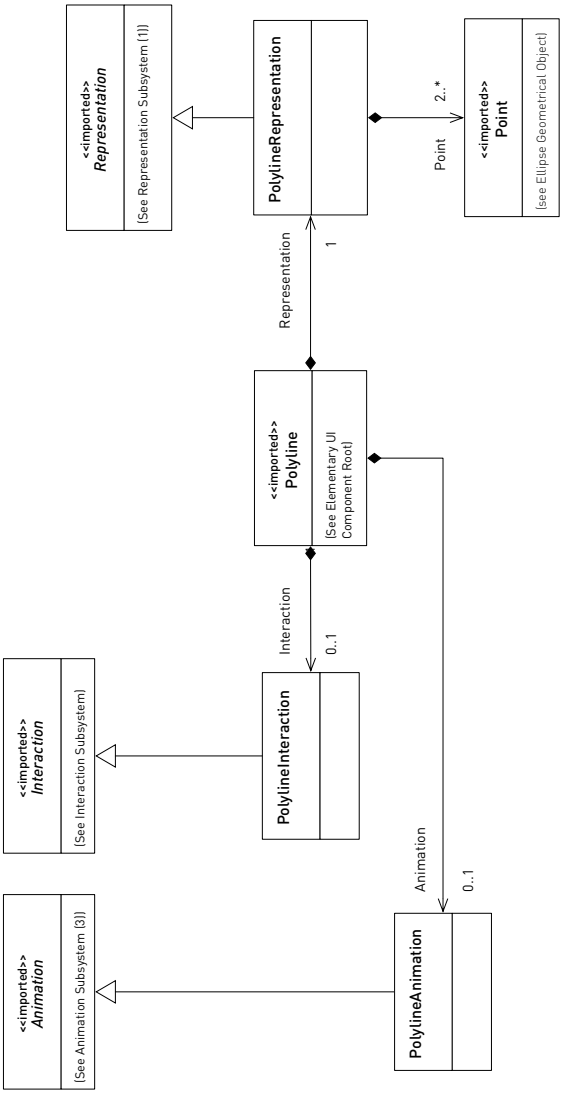


Figure B.50: Core Language Model: Polyline Geometrical Object.

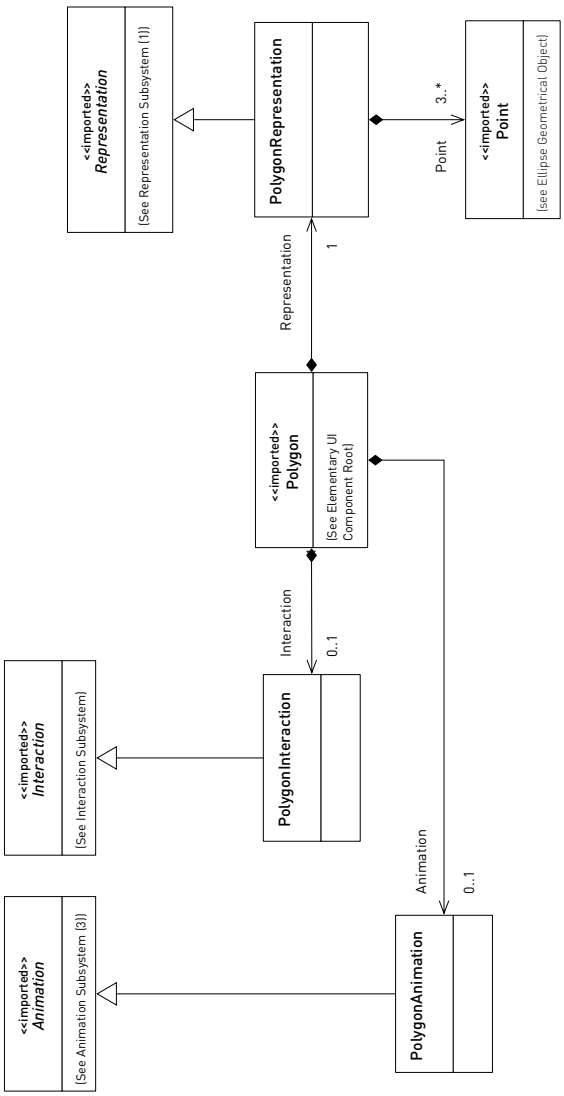


Figure B.51: Core Language Model: Polygon Geometrical Object.

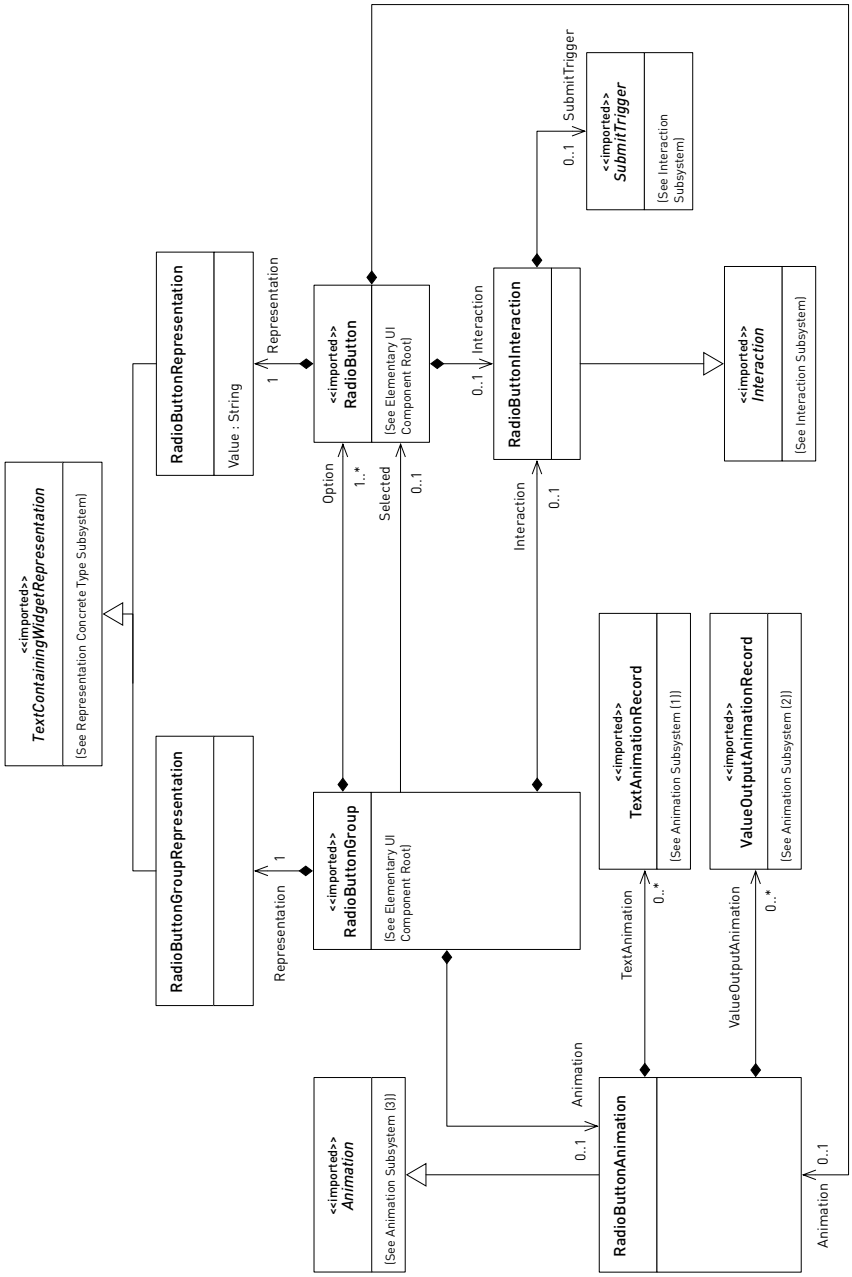


Figure B.52: Core Language Model: Radio Button Widget.

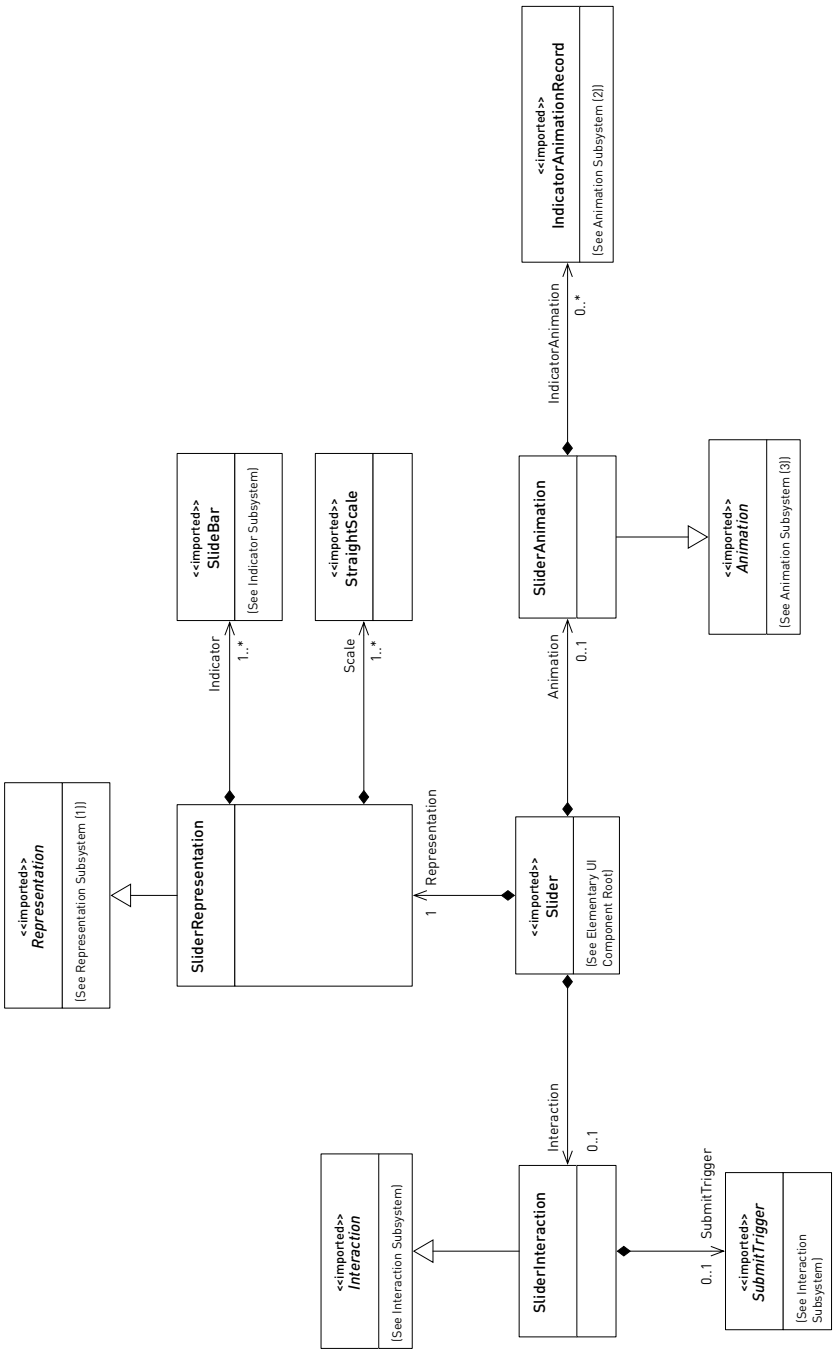


Figure B.53: Core Language Model: Slider Widget.

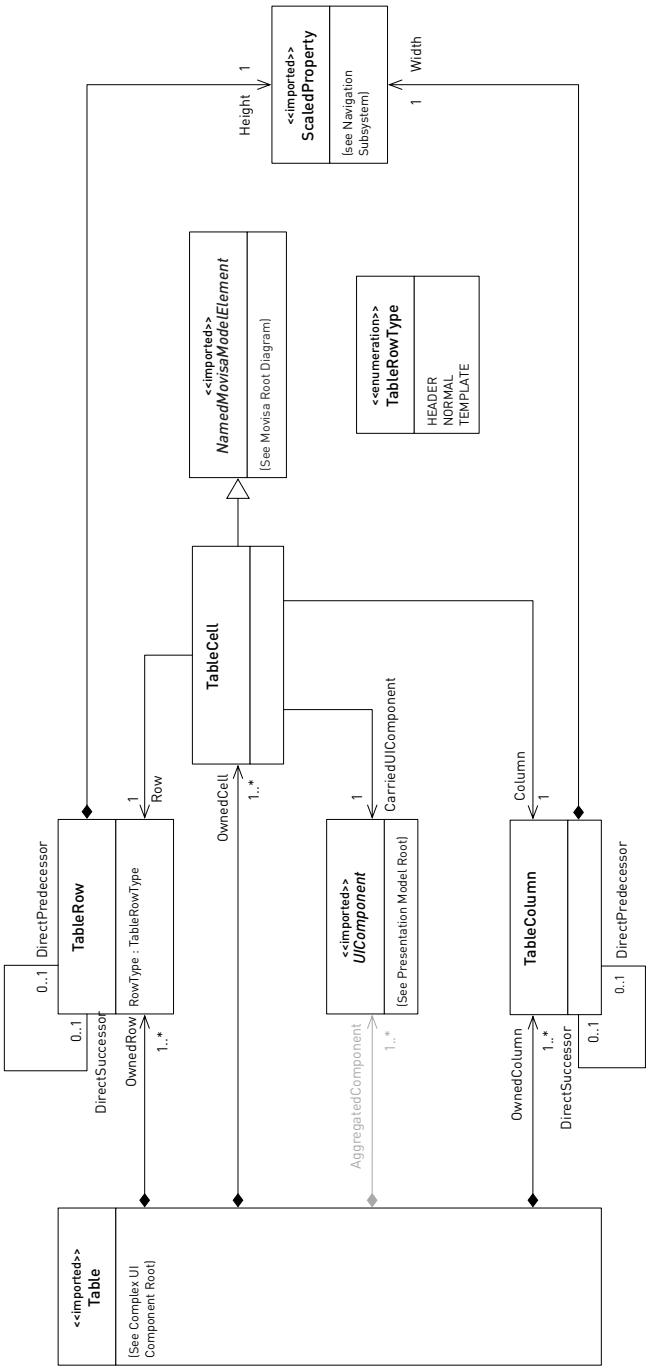


Figure B.54: Core Language Model: Table Widget (Structure).

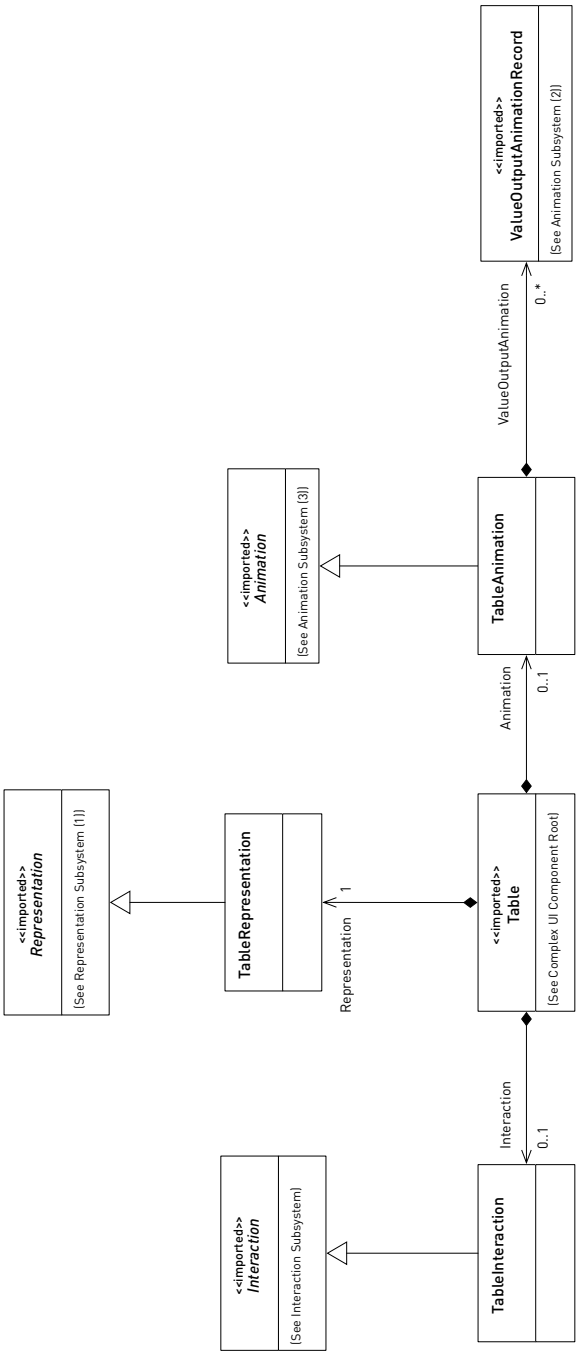


Figure B.55: Core Language Model: Table Widget (Properties).

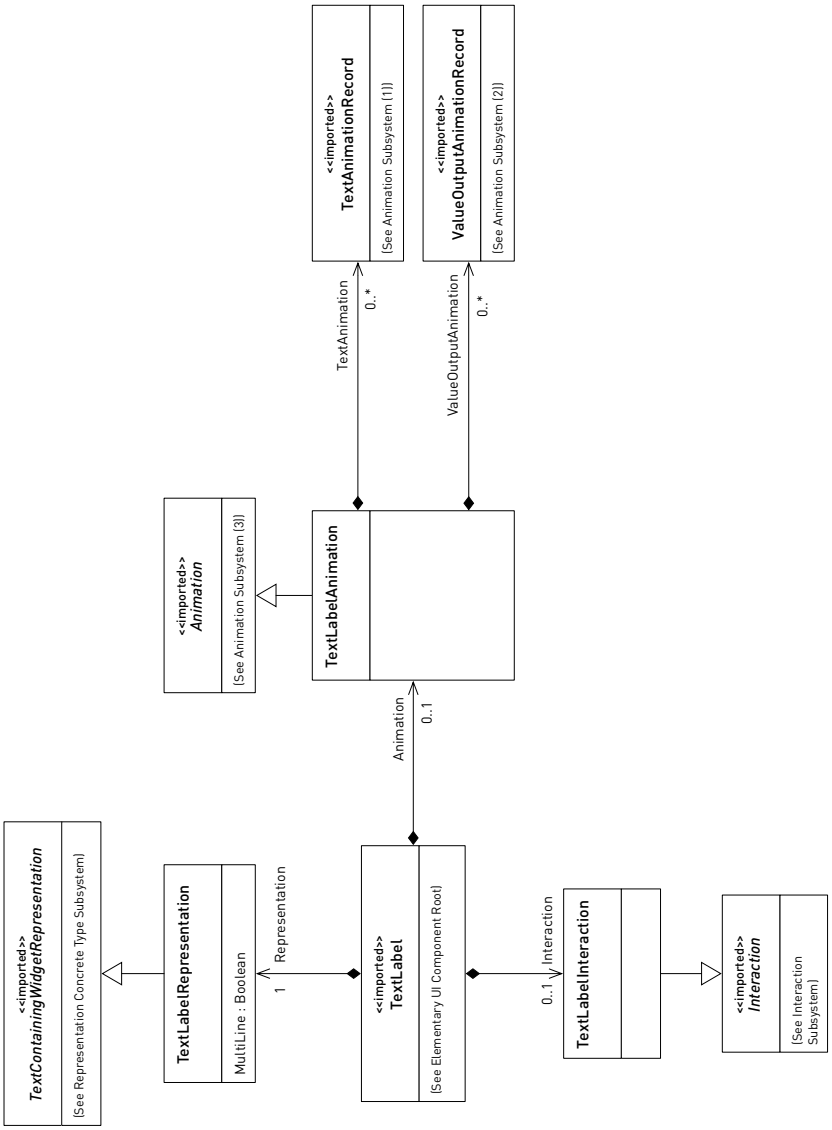


Figure B.56: Core Language Model: Text Label Widget.

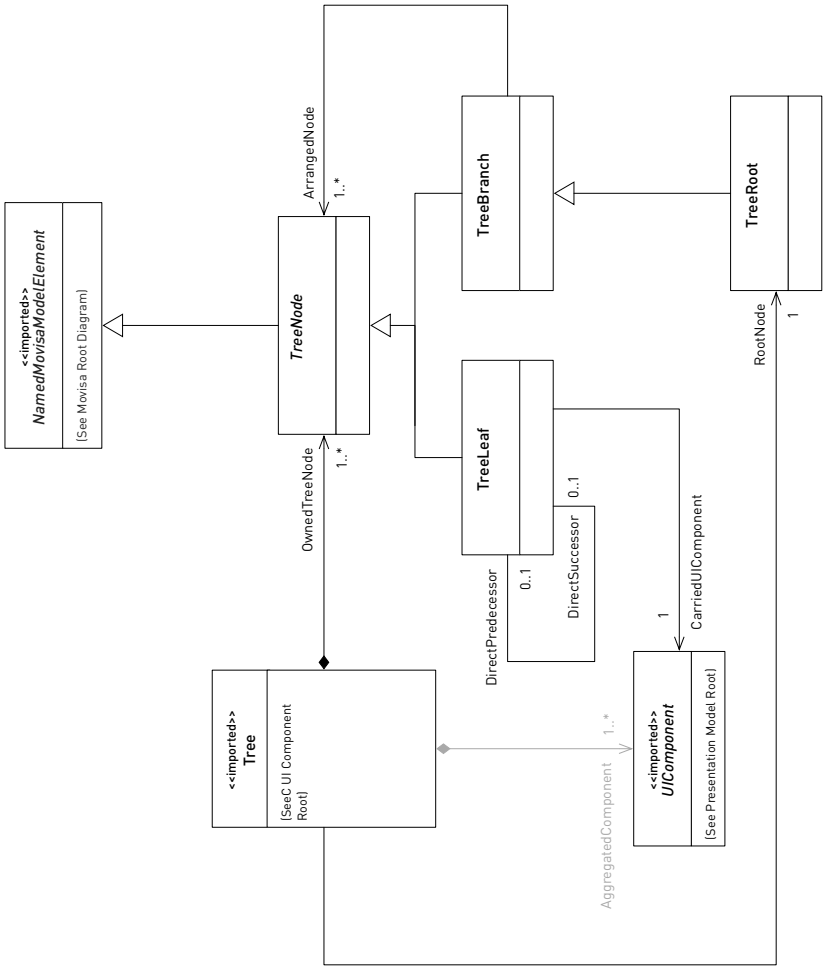


Figure B.57: Core Language Model: Tree Widget (Structure).

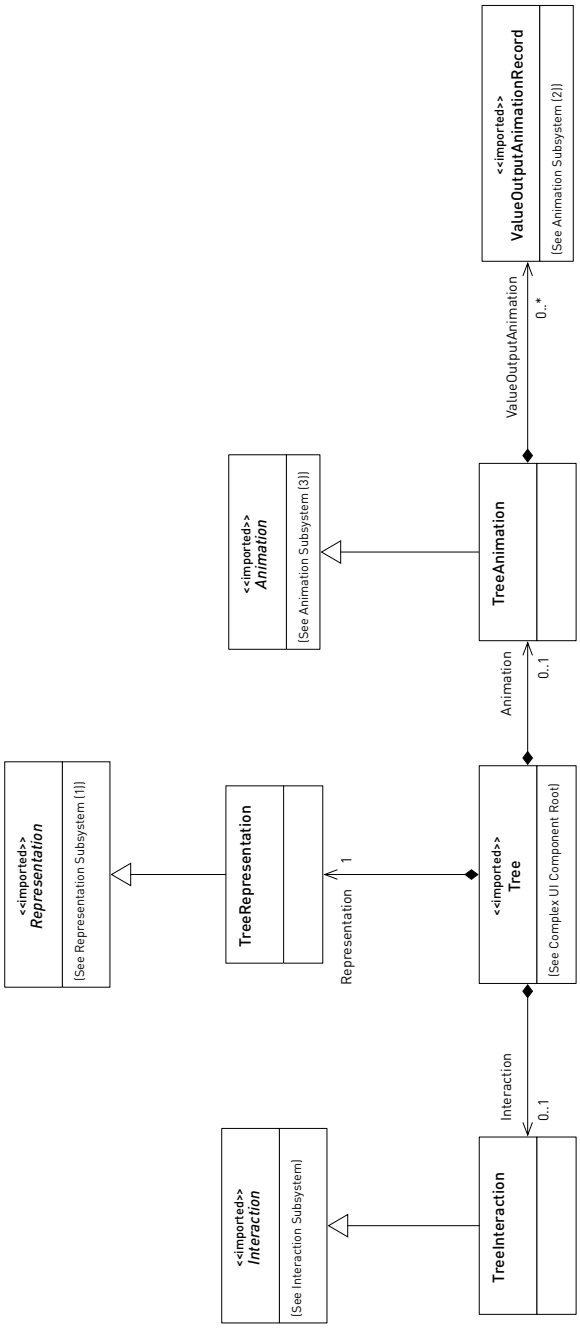


Figure B.58: Core Language Model: Tree Widget (Properties).

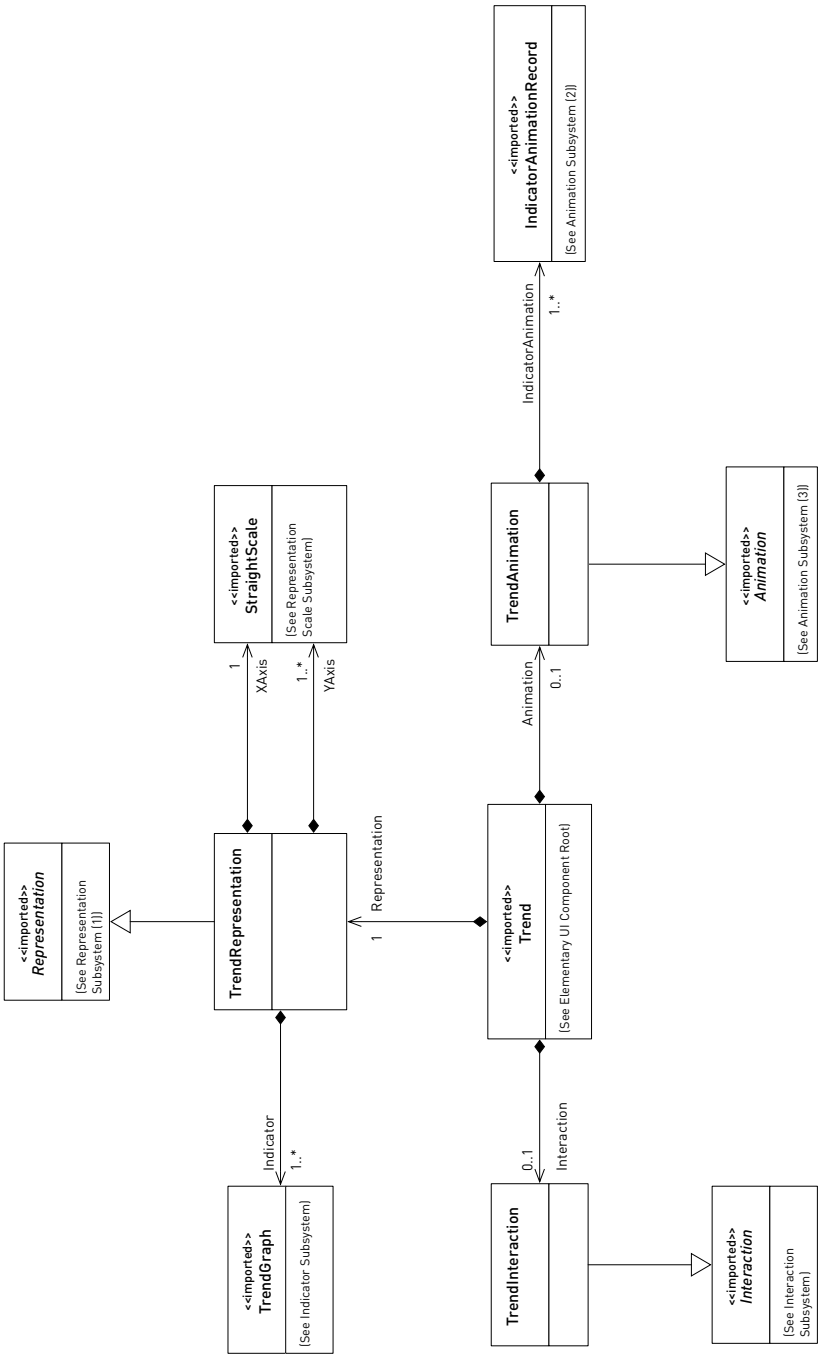


Figure B.59: Core Language Model: Trend Widget.

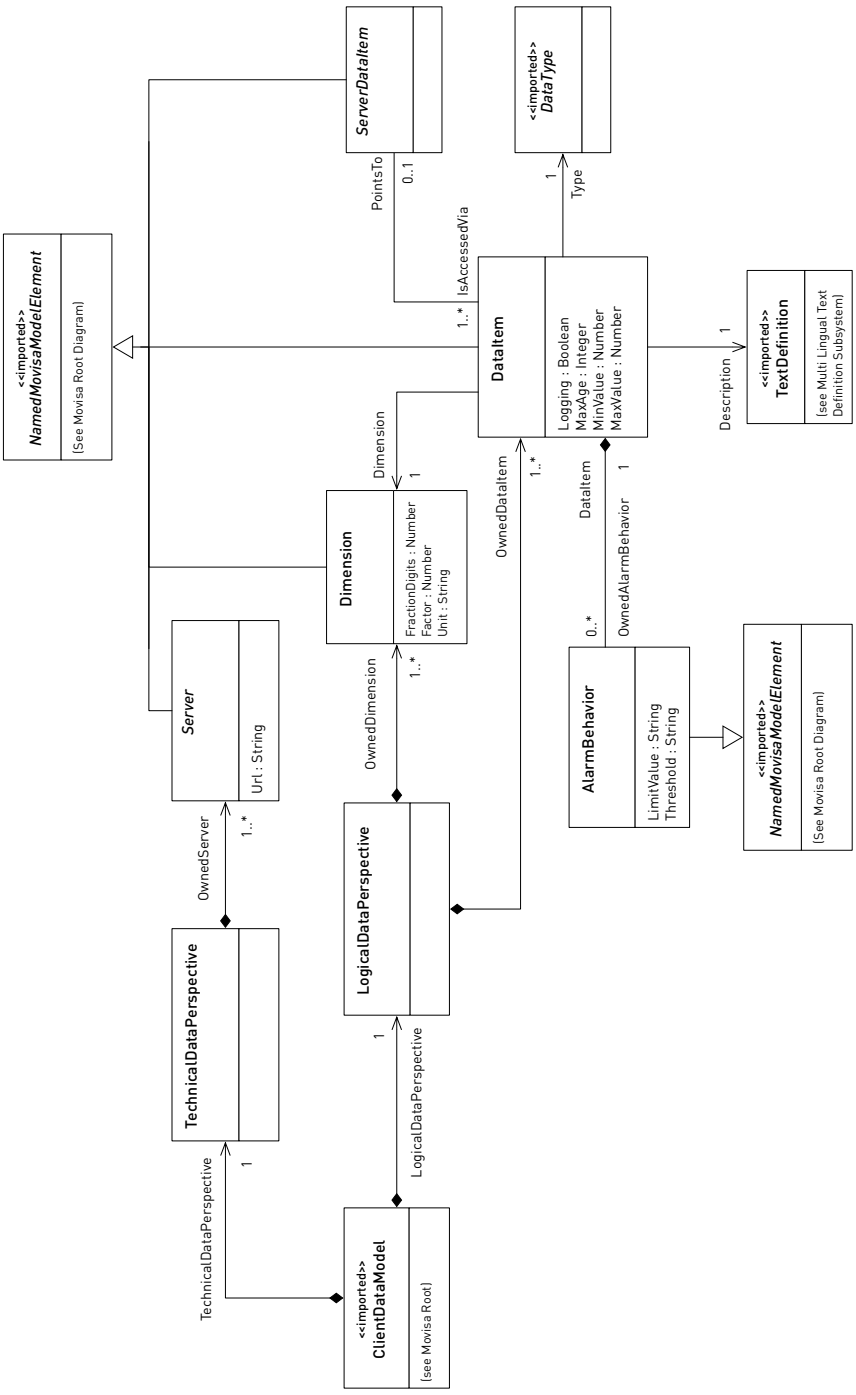


Figure B.60: Core Language Model: Client Data Model Root.

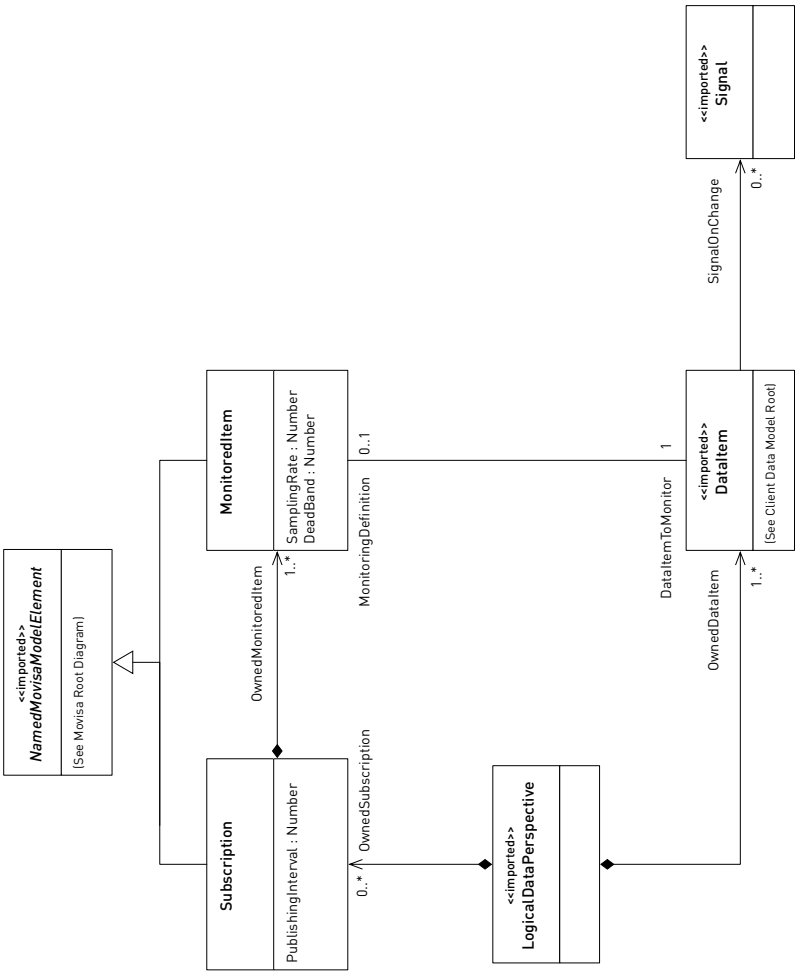


Figure B.61: Core Language Model: Logical Data Perspective.

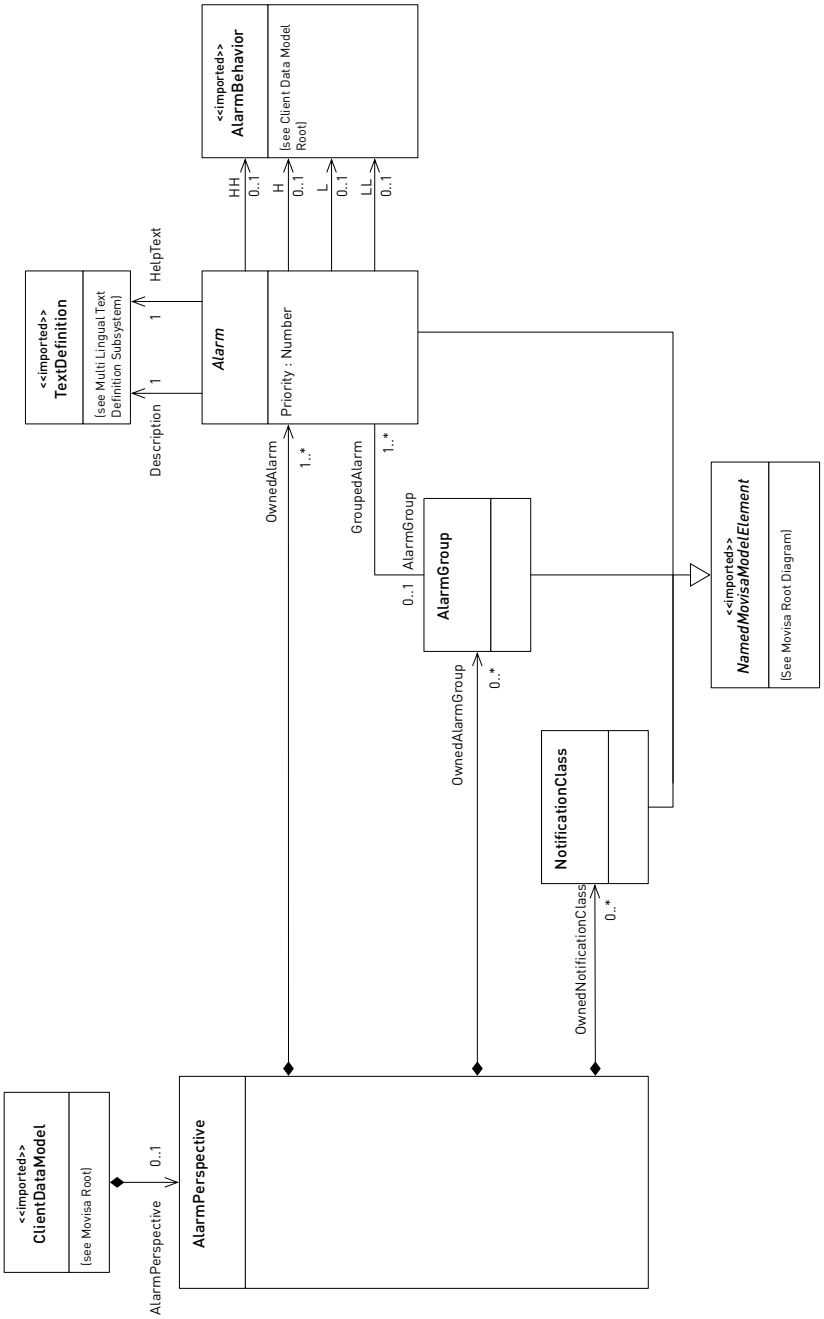


Figure B.62: Core Language Model: Alarm Perspective (1).

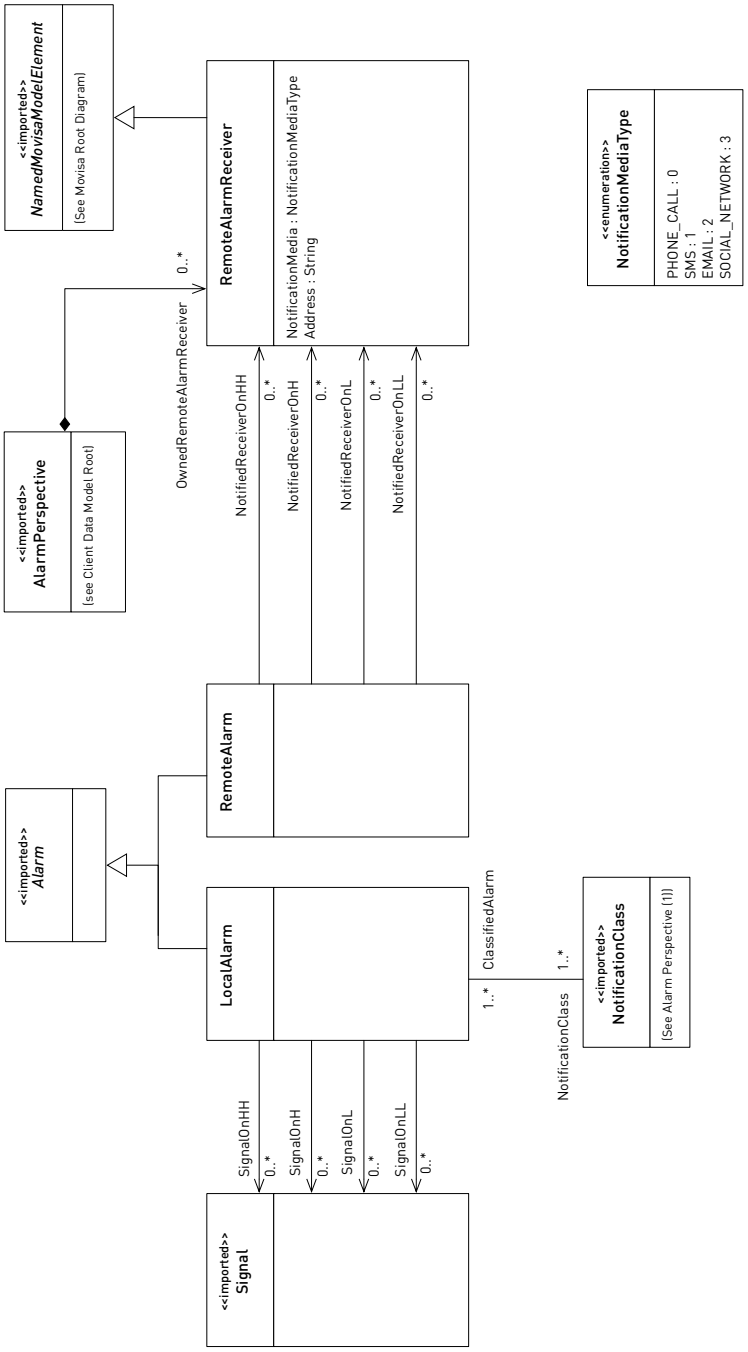


Figure B.63: Core Language Model: Alarm Perspective (2).

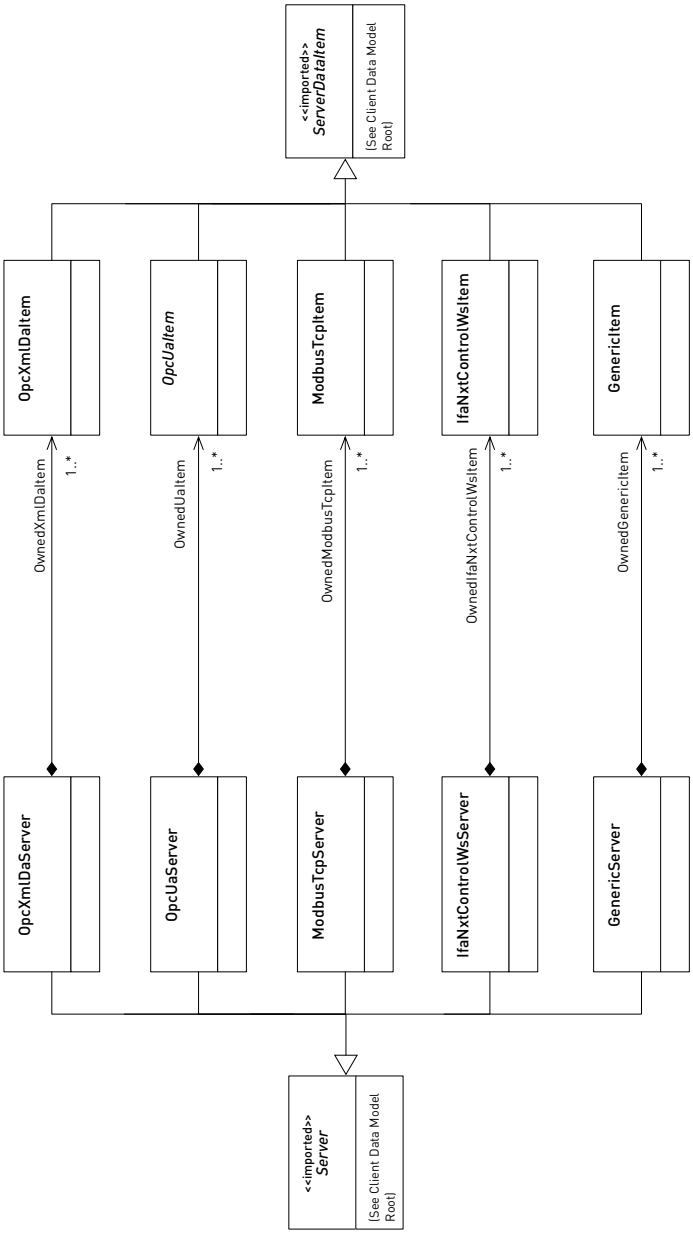


Figure B.64: Core Language Model: Technical Data Perspective Root.

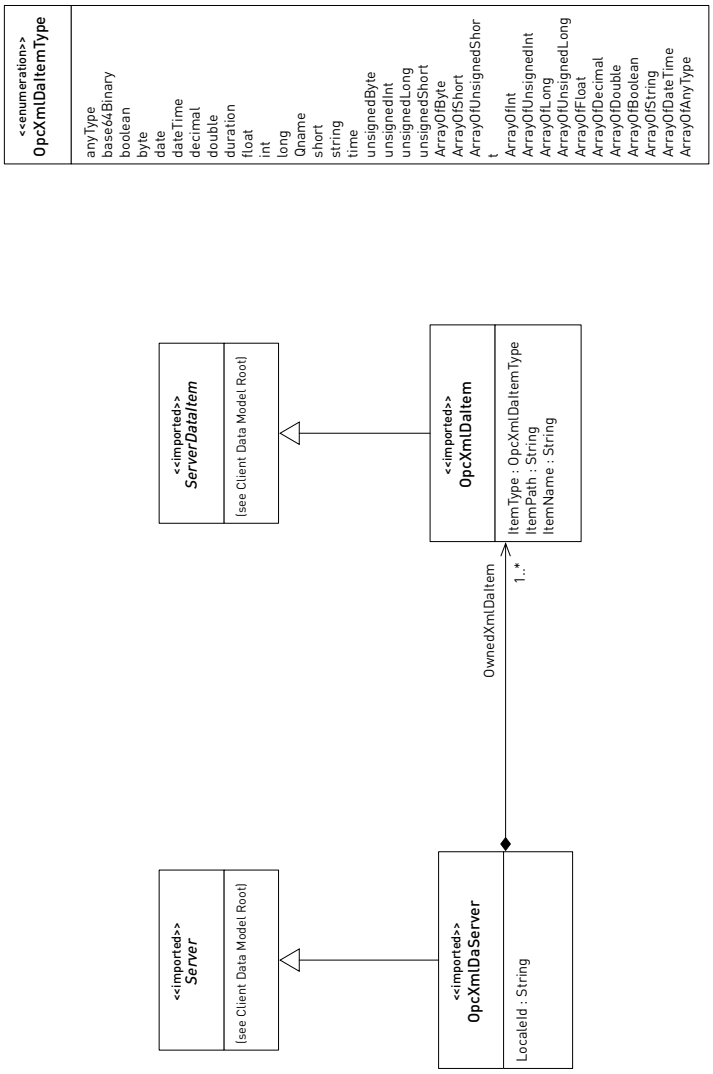


Figure B.65: Core Language Model: Technical Data Perspective (OPC XML-DA).

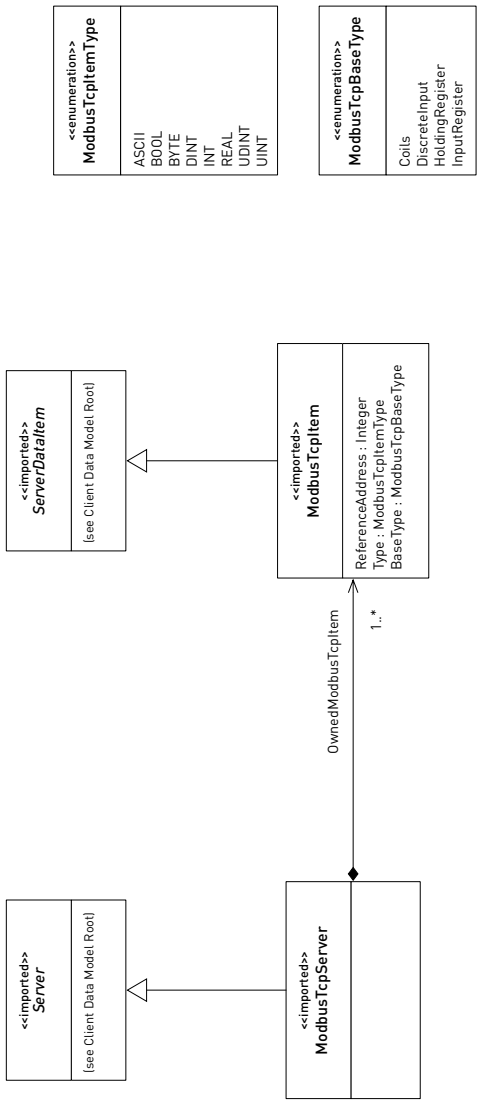


Figure B.66: Core Language Model: Technical Data Perspective (Modbus TCP).

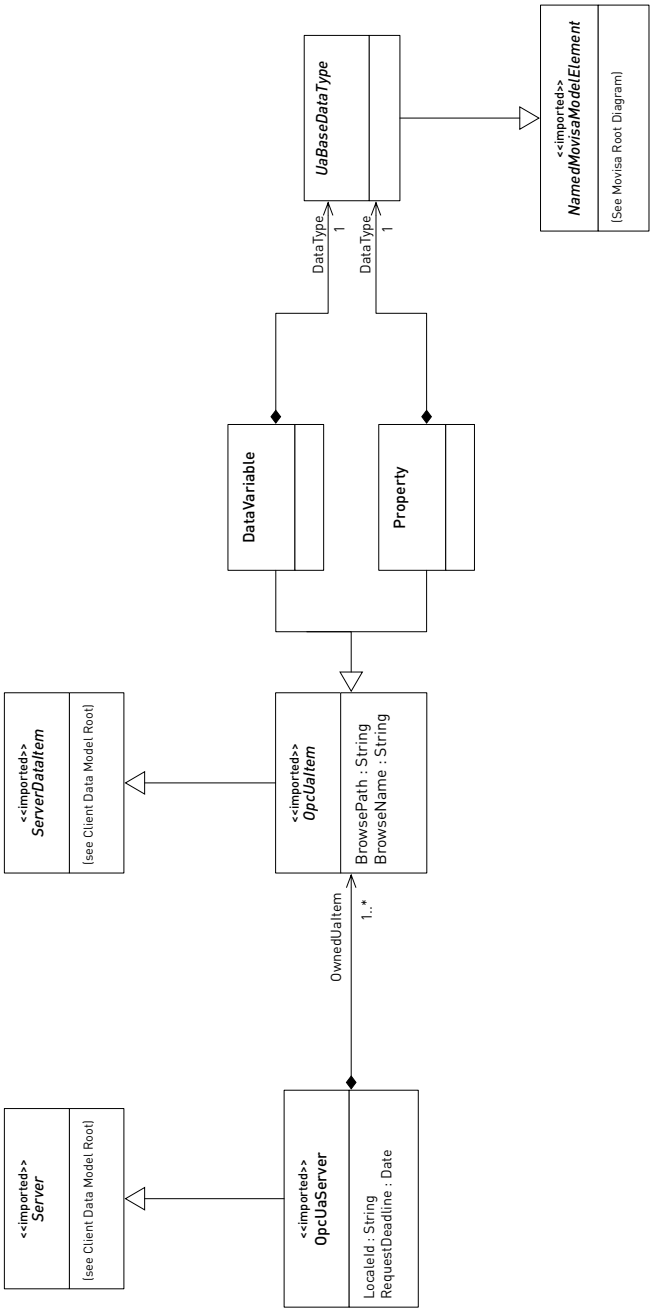


Figure B.67: Core Language Model: Technical Data Perspective (OPC UA).

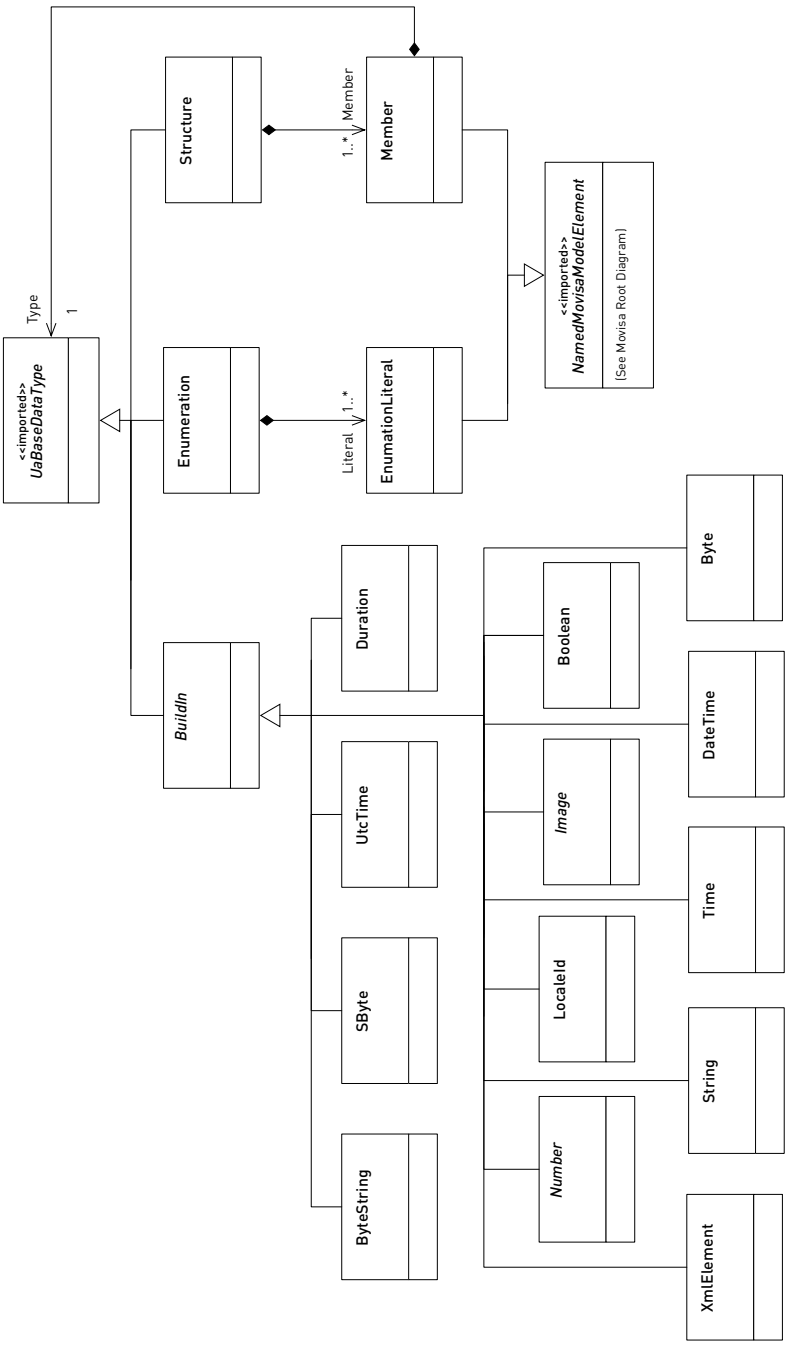


Figure B.68: Core Language Model: Technical Data Perspective (OPC UA, Data Types 1).

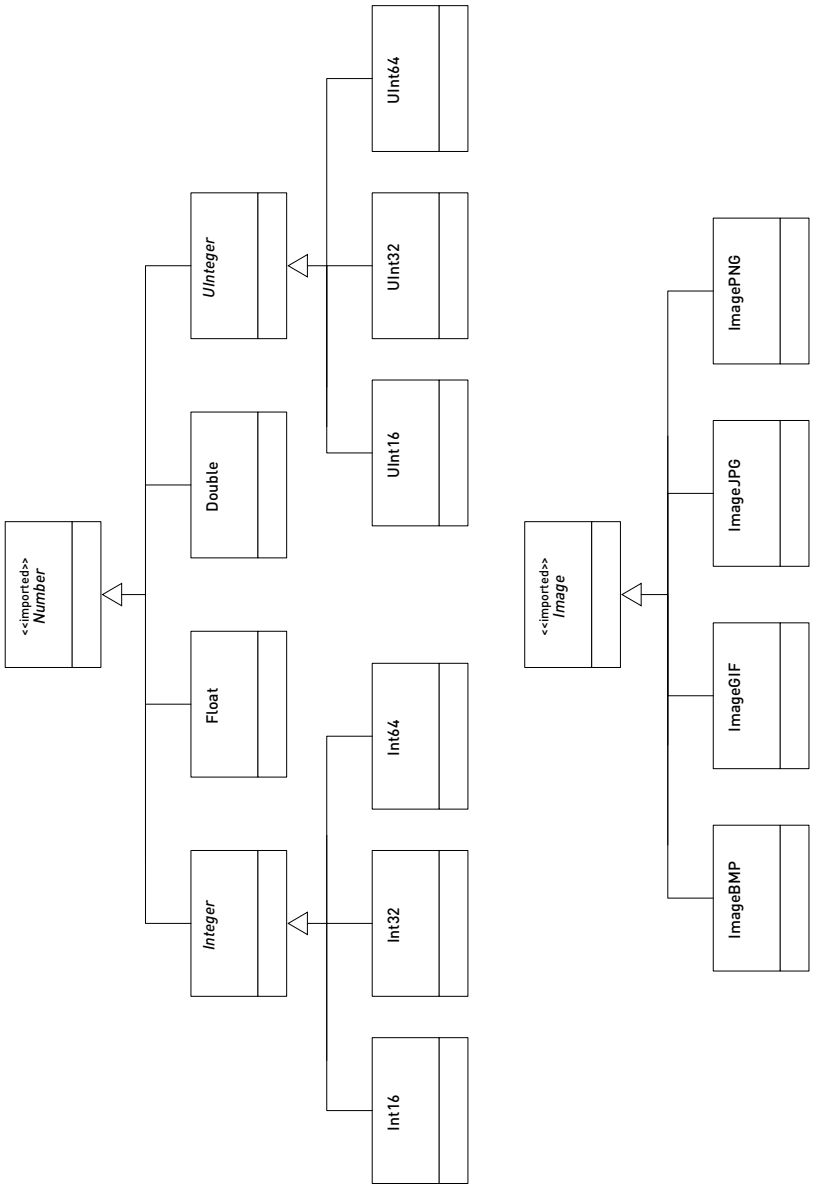


Figure B.69: Core Language Model: Technical Data Perspective (OPC UA, Data Types 2).

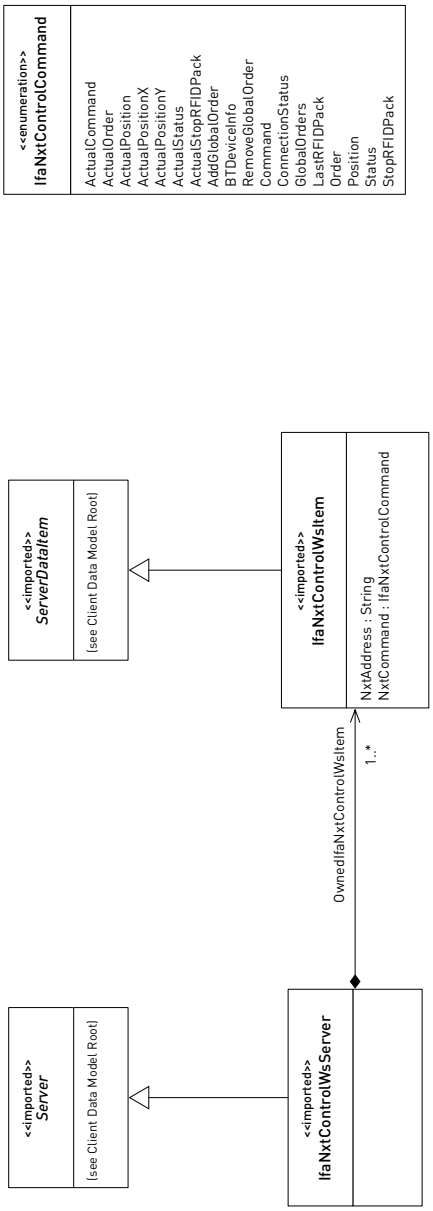


Figure B.70: Core Language Model: Technical Data Perspective (IfaNxtControlWs).

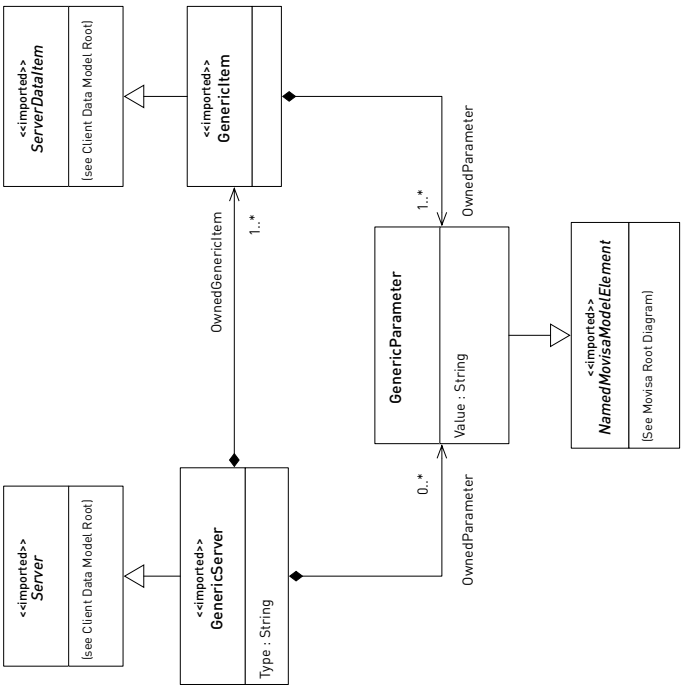


Figure B.71: Core Language Model: Technical Data Perspective (Generic Server).

Appendix C

Case Study Results

This chapter presents screenshots of the visualization solutions that have been modeled and generated in Chapter 6.

C.1 Process Industries Visualization Solution

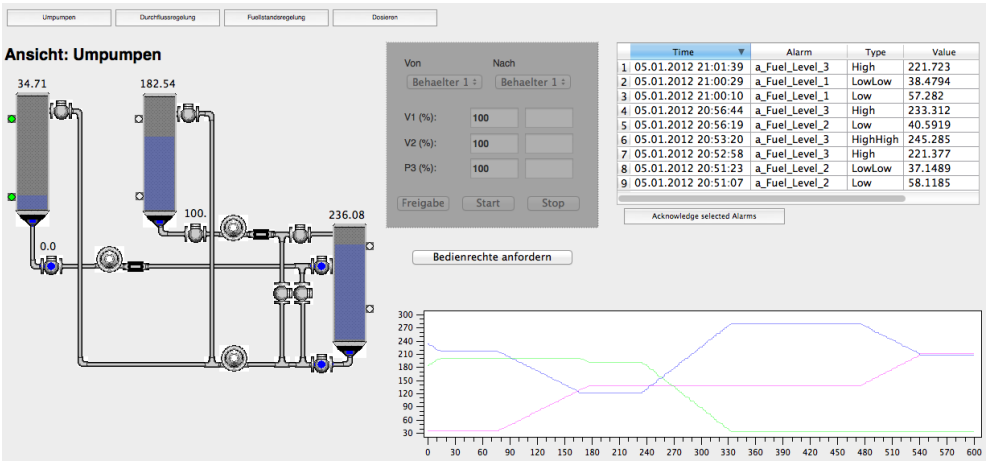


Figure C.1: Process Industries Case Study: Generated *Python* based runtime solution (non-sandboxed). It can be seen that the operating faceplate is blocked. To demonstrate multilingualism, the user interface language was switched to *German*.

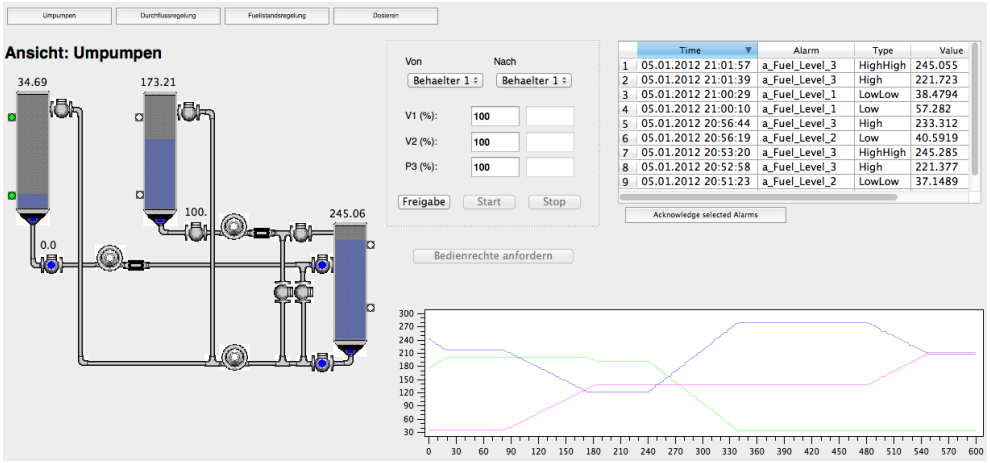


Figure C.2: Process Industries Case Study: Generated *Python* based runtime solution (non-sandboxed). It can be seen that the operating faceplate is released and, thus, interventions in the process are allowed.

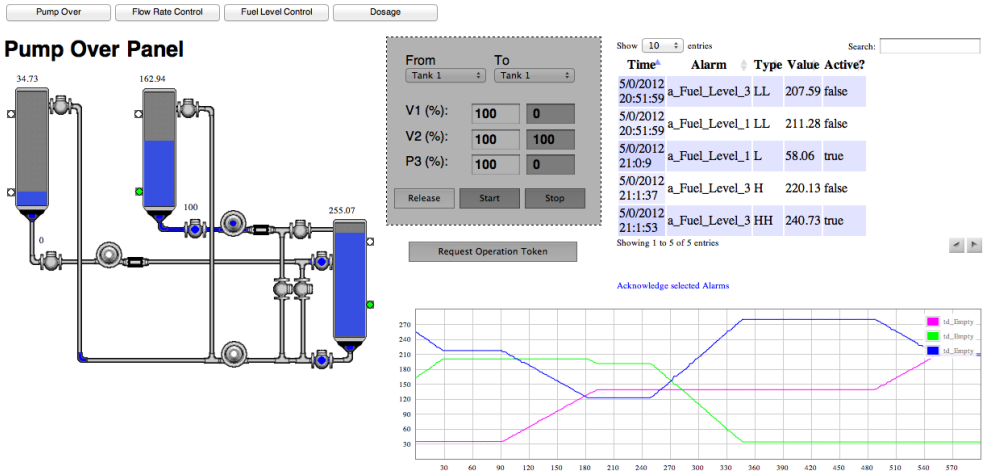


Figure C.3: Process Industries Case Study: Generated *HTML* based runtime solution (sandboxed). It can be seen that the operating faceplate is blocked. Additionally, the button to request the operation token is also blocked due to interventions by other operators (e.g. through the visualization solution shown in Figure C.2). To demonstrate multilingualism, the user interface language was switched to *English*.

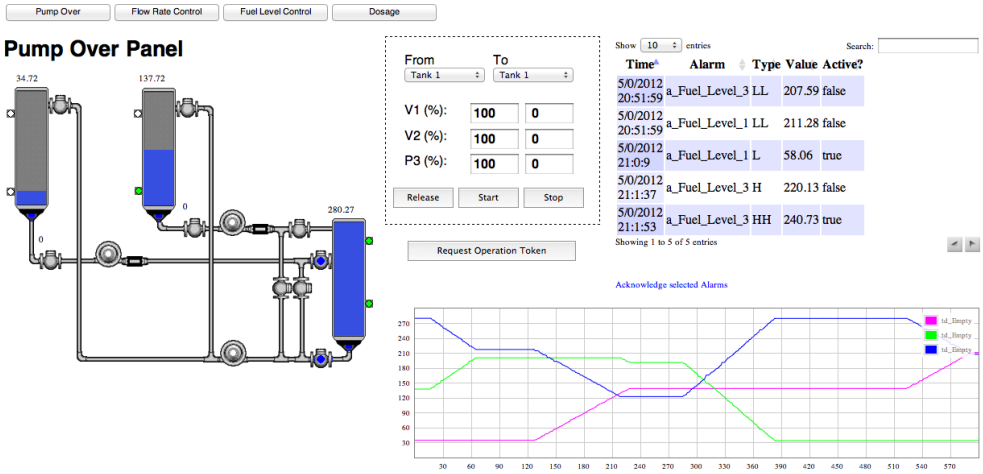


Figure C.4: Process Industries Case Study: Generated *HTML* based runtime solution (sandboxed). It can be seen that the operating faceplate is released and, thus, interventions in the process are allowed.

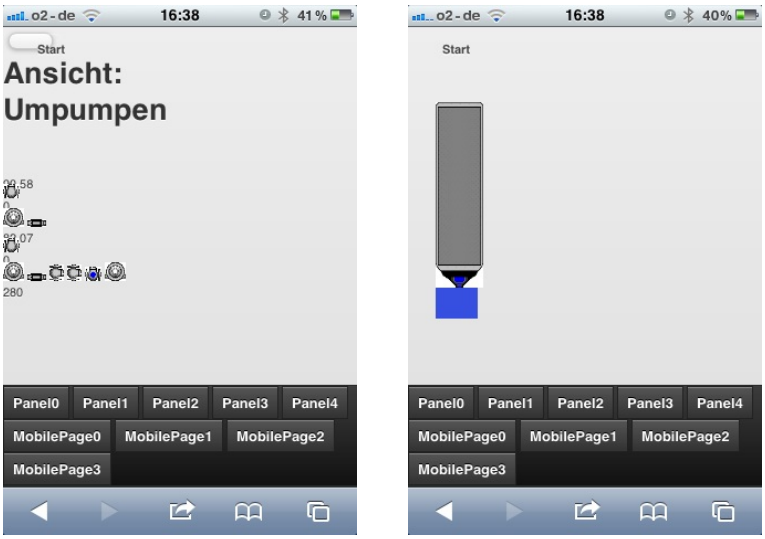


Figure C.5: Process Industries Case Study: After applying a horizontal CUI-to-CUI transformation and a subsequent vertical CUI-2-FUI transformation, the resulting solution can be used on the iPhone. These screenshots show the solution without manually improving the model after translating it into this new context of use.

C.2 Factory Automation Visualization Solution

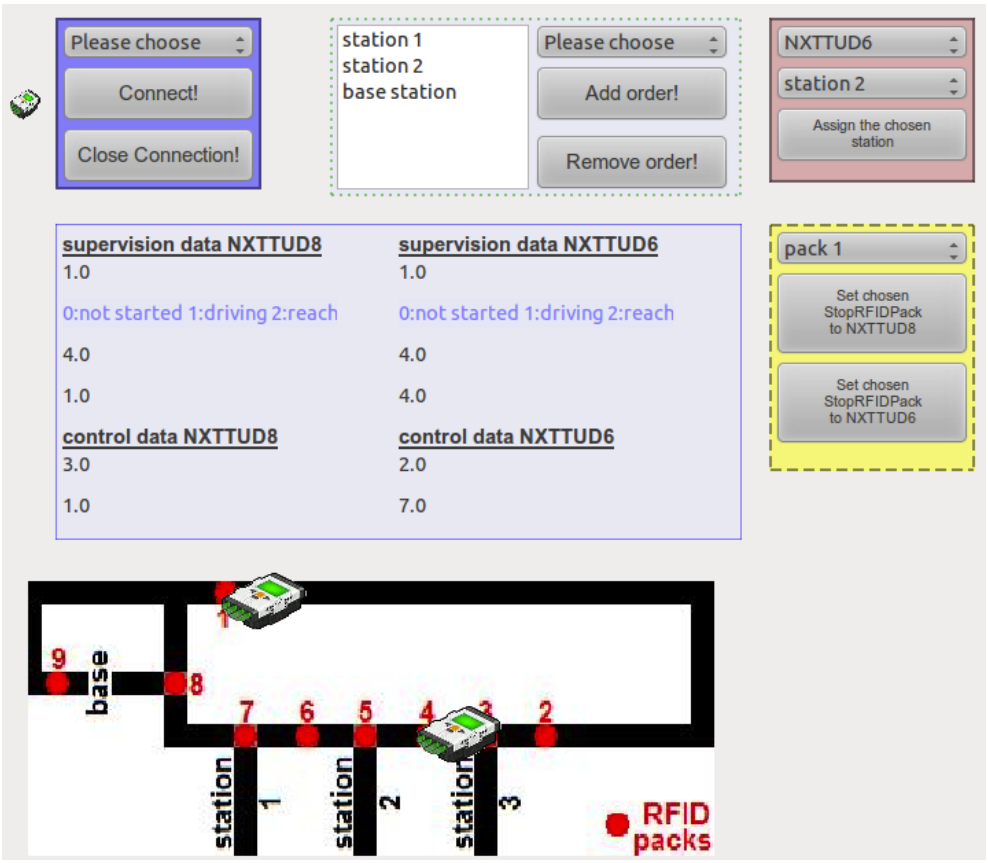


Figure C.6: Factory Automation Case Study: Generated *Python* based runtime solution (non-sandboxed).

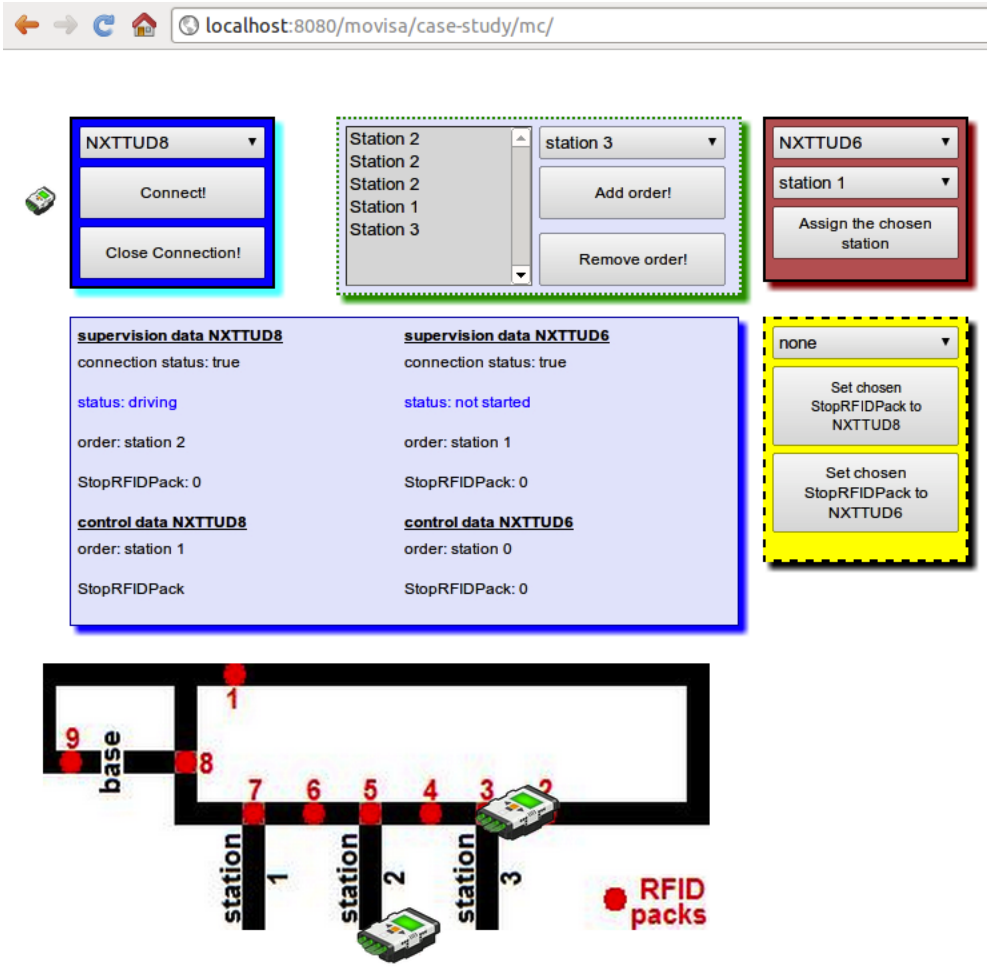


Figure C.7: Factory Automation Case Study: Generated *HTML* based runtime solution (sandboxed).

C.3 Energy Supply System Visualization Solution

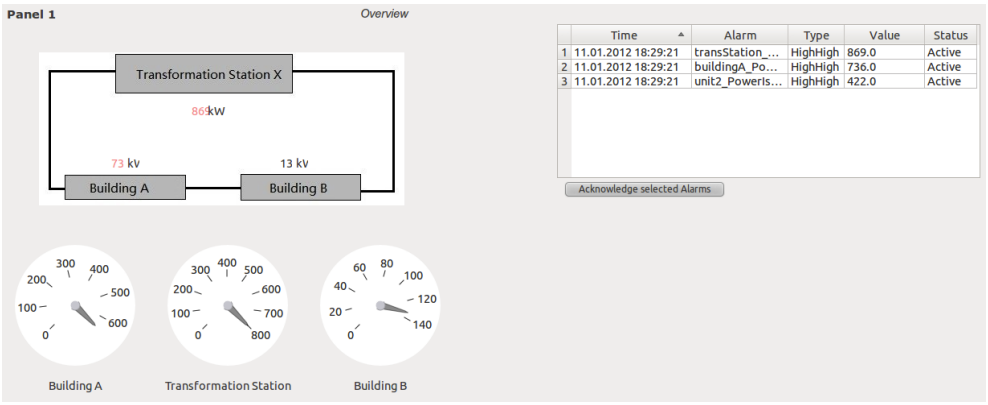


Figure C.8: Energy Supply System Case Study: Generated *Python* based runtime solution (non-sandboxed). This figure presents the topmost hierarchy level of the energy supply network. It can be seen that alarms were thrown indicating that the network has a too high load.

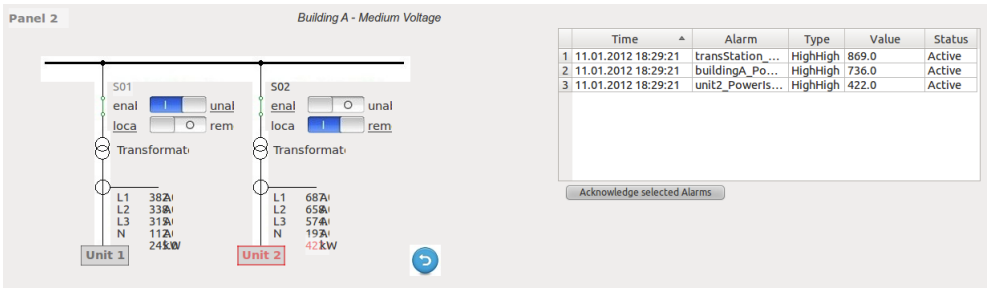


Figure C.9: Energy Supply System Case Study: Generated *Python* based runtime solution (non-sandboxed). This figure presents a view on the energy supply network of "Building A". It can be seen that the reason for the alarms is located in "Unit 2".

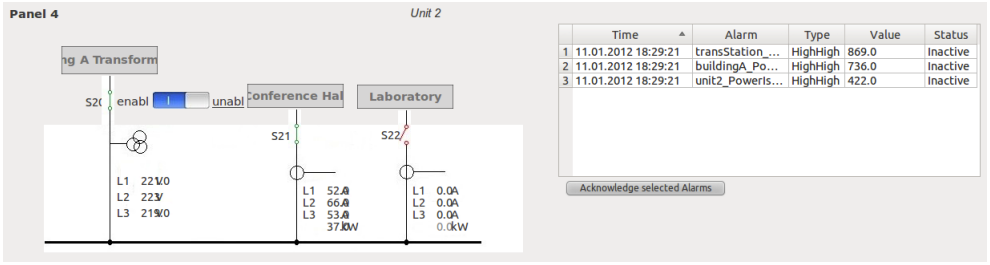


Figure C.10: Energy Supply System Case Study: Generated *Python* based runtime solution (non-sandboxed). This figure presents a view on the energy supply network of "Unit 2" in "Building A". The reason for the alarms could be identified and eliminated. Hence, the alarms are no longer in *active* state, even though they are *not acknowledged*.

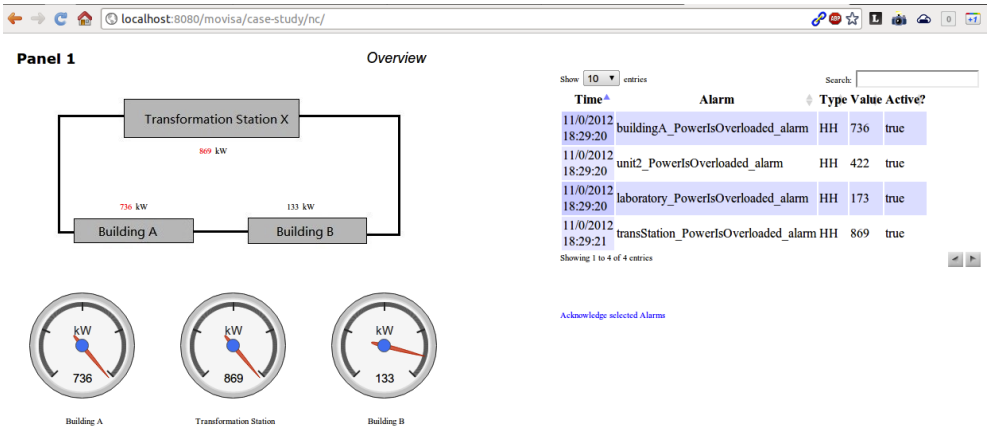


Figure C.11: Energy Supply System Case Study: Generated *HTML* based runtime solution (sandboxed). This figure presents the topmost hierarchy level of the energy supply network. It can be seen that alarms were thrown indicating that the network has a too high load.

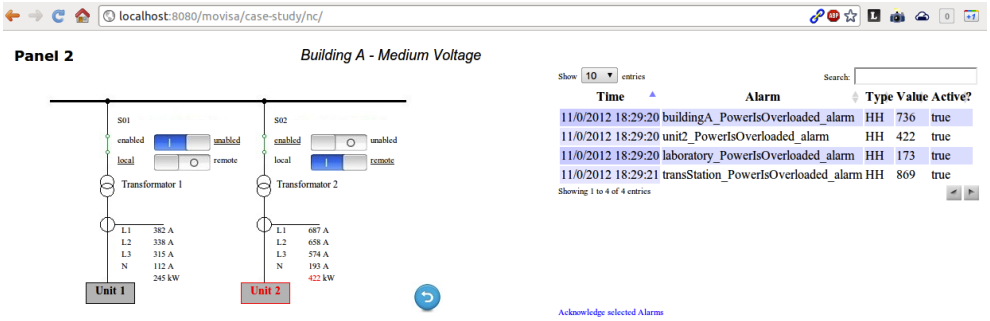


Figure C.12: Energy Supply System Case Study: Generated *HTML* based runtime solution (sandboxed). This figure presents a view on the energy supply network of "Building A". It can be seen that the reason for the alarms is located in "Unit 2".

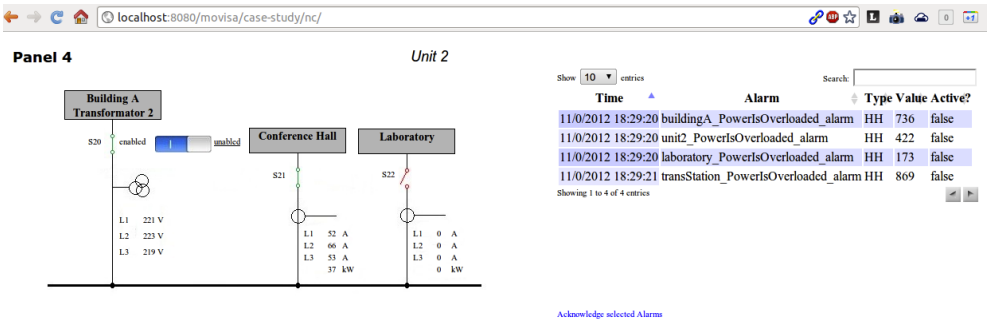


Figure C.13: Energy Supply System Case Study: Generated *HTML* based runtime solution (sandboxed). This figure presents a view on the energy supply network of "Unit 2" in "Building A".The reason for the alarms could be identified and eliminated. Hence, the alarms are no longer in *active* state, even though they are *not acknowledged*.

C.4 Health Care Visualization Solution

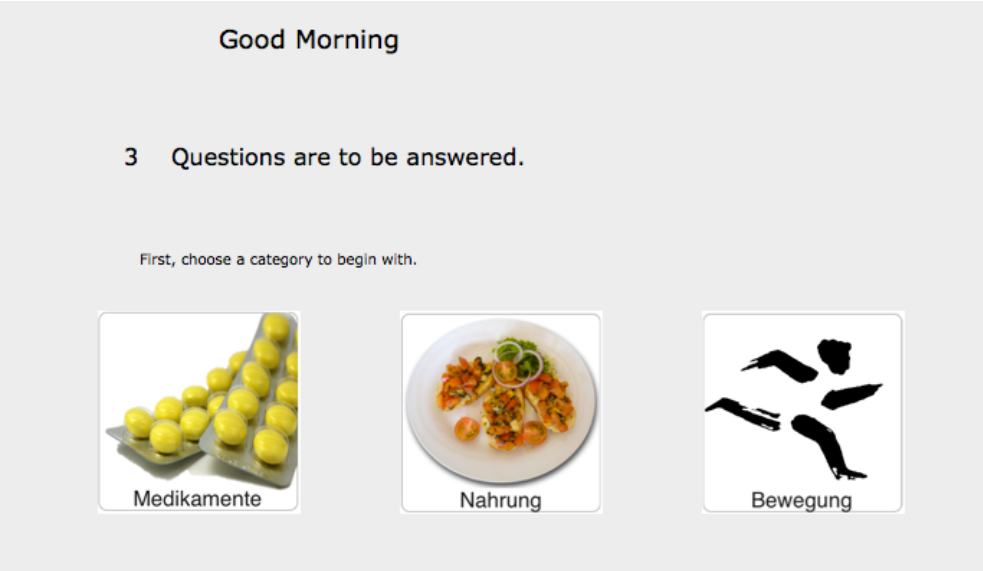


Figure C.14: Health Care Case Study: Generated *Python* based runtime solution (non-sandboxed). Patients has to answer questions about their recent habits.

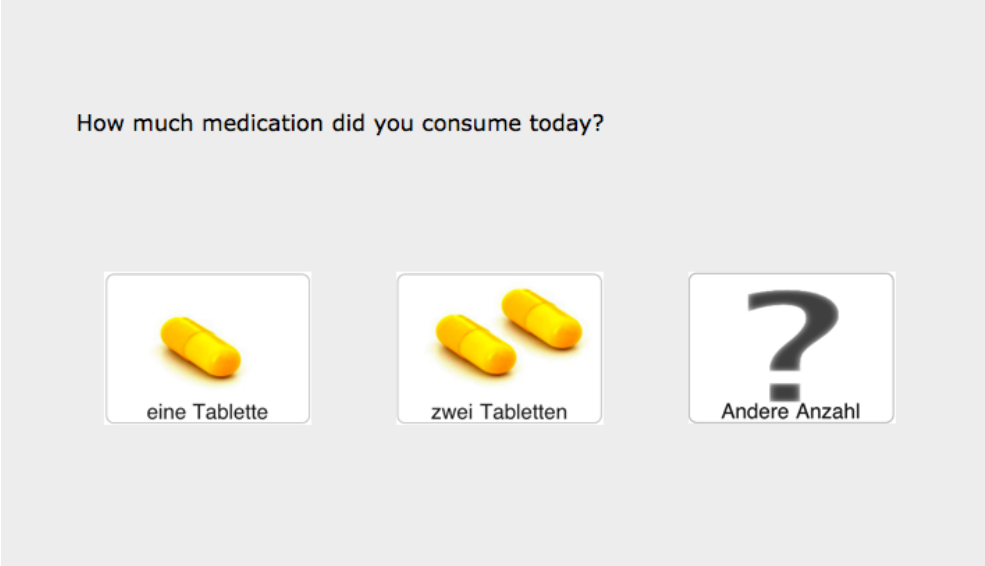


Figure C.15: Health Care Case Study: Generated *Python* based runtime solution (non-sandboxed). First, a patient decided to fill in the number of tablets recently consumed.

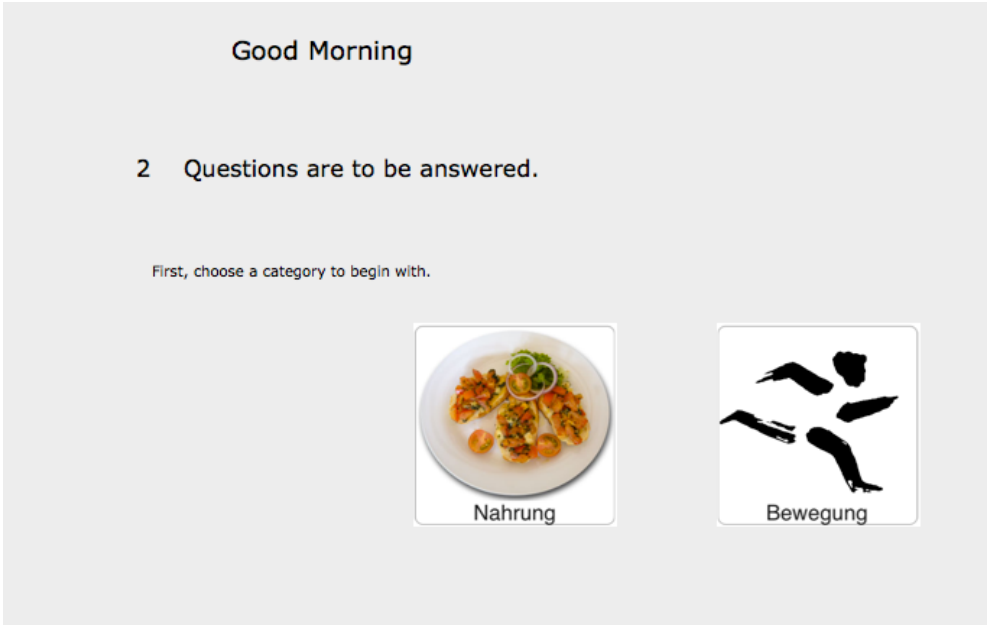


Figure C.16: Health Care Case Study: Generated *Python* based runtime solution (non-sandboxed). After providing the answer about medication, this option disappeared from the screen and, thus, it cannot be selected no longer.

Stefan Hennig

born on June 27th, 1980
in Lutherstadt Wittenberg, Germany

Curriculum Vitae

Career

11/2009 – 04/2012	**Doctoral Studies** at the Institute of Automation, Technische Universität Dresden *"Landesinnovationspromotion" funded by the European Social Fund and the Freestate of Saxony*
10/2007 – 04/2012	**Research Associate** at the Institute of Automation, Technische Universität Dresden
09/2006 – 09/2007	**Research Assistant** at the Institute of Automation, Technische Universität Dresden

Studies

10/2000 – 08/2006	**Studies of Information Systems Engineering** at the *Technische Universität Dresden*: Interdisciplinary course of the Faculties *Electrical and Computer Engineering* and *Computer Science* Graduation: **Diplomingenieur** (Master of Science)

School Education

1992 – 1999	**Albert-Schweitzer-Gymnasium Coswig/Anhalt** Graduation: **Abitur** (Higher Education)
1987 – 1992	**Polytechnische Oberschule Klieken**