Application-specific UML Profiles for Multidisciplinary Product Data Integration

A thesis accepted by the Faculty of Aerospace Engineering and Geodesy of the Universität Stuttgart in partial fulfilment of the requirements for the degree of Doctor of Engineering Sciences (Dr.-Ing.)

by
Dipl.-Ing. Axel Reichwein
born in Cologne, Germany

Main referee:	Priv.Doz. Dr.-Ing. Stephan Rudolph
Co-referee:	Prof. Dr.-Ing. habil. Bernd Kröplin
Co-referee:	Prof. Dr.-Ing. Reinhard Reichel
Date of defence:	December 6, 2011

Institute of Statics and Dynamics of Aerospace Structures
Universität Stuttgart
2011

Contents

-

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Stephan Rudolph for giving me the opportunity to learn, grow, and explore. The achievement of this work was greatly influenced by his vision and the work environment he set up in our research group. His passionate interest in research and his constant energetic enthusiasm were always very motivating and allowed me to start this thesis with great confidence. Obviously, the initial fun and excitement was quickly replaced by challenging questions and headaches. But Dr. Stephan Rudolph's guidance and open-mindedness as well as the numerous interactions with the group members were very helpful in overcoming many difficulties.

In addition, I would like to extend my thanks to my committee members Prof. Bernd Kröplin and Prof. Reinhard Reichel for their valuable support and interest.

Among the group members, I would especially like to express my gratitude to Dr. Peter Hertkorn with whom I worked closely. His expertise in computer science and his readiness to share it without presumption were extremely beneficial. I also thank other group members including Michael Bölling, Peter Arnold, Johannes Gross, Marc Eheim and Martin Motzer for contributing to a very friendly group atmosphere in which we could not only discuss serious academic topics but also enjoy entertaining philosophical debates over lunch.

Finally, I am grateful to my parents for their continuous support and for believing in me throughout the many challenges.

-

Abbreviations

AEC	Architecture, Engineering and Construction
AFDX	Avionics Full-Duplex Switched Ethernet
AOCS	Attitude and Orbit Control System
AP	Application Protocol
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BOM	Bill of Materials
CAD	Computer-Aided Design
CAE	Computer-Aided Engineering
CAN	Controller Area Network
CC	Conformance Classes
CCP	Cut Copy and Paste
CIMsteel	Computer Integrated Manufacturing of Constructional Steelwork
CIS/2	CIMsteel Integration Standard Version2
COMBINE	Computer Models for the Building Industry in Europe
CPIOM	Core Processing Input/Output Module
DAML	DARPA Agent Markup Language
DL	Description Logics
FM	Facility Management
HVAC	Heating, Ventilating and Air Conditioning
IDE	Integrated Development Environment
IFC	Industry Foundation Classes

IMA	Integrated Modular Avionics
ISO	International Organization for Standardization
IGES	Initial Graphics Exchange Specification
KIF	Knowledge Interchange Format
MDA	Model Driven Architecture
MOF	Meta Object Facility
OCL	Object Constraint Language
OCSM	Outflow Valve Control and Sensor Module
OIL	Ontology Interchange Language
OMG	Object Management Group
ORVD	Outflow Relief Valve Dumps
OWL	Web Ontology Language
PDM	Product Data Management
PID	Proportional-Integral-Derivative
PIM	Platform Independent Model
PLM	Product Lifecycle Management
PPT	Pulsed Plasma Thruster
PSM	Platform Specific Model
QVT	Query/View/Transformation
RDF	Resource Description Framework
RDFS	RDF Schema
SDAI	Standard Data Access Interface
SPG	Solution Path Generator
SysML	Systems Modeling Language
SWRL	Semantic Web Rule Language
STEP	Standard for the Exchange of Product Model Data
UML	Unified Modeling Language
URL	Unified Resource Locator

VB	Visual Basic
VBS	Visual Basic Script
VBA	Visual Basic Application
VRML	Virtual Reality Modeling Language
W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Abstract

This thesis examines the suitability of the Unified Modeling Language (UML) to establish a central product model for multidisciplinary product data integration. Computer-aided product design involves the use of specialized discipline-specific software applications in order to model and simulate various product aspects. Dependencies between models are thereby frequent as the same product information often appears redundantly in various engineering models. In addition, dependencies exist due to relationships between distinct features of various models. As a result, model modifications frequently require the update of dependent models. Data consistency between models is achieved automatically through model-to-model data exchange software.

The use of a central product model enables to reduce the required number of data exchange connections. Central product models store product information which is spread across several models and achieve data consistency through data exchange connections between themselves and specific models as in a hub-and-spoke network. Central product models are especially useful for automatic data consistency in design scenarios which include a high number of inter-model dependencies and model modifications.

The integration of geometry and therefrom derived models such as structural analysis or computational fluid dynamics models has already been successfully addressed in numerous central product models. However, the multidisciplinary integration of more diverse models, such as geometric, software, controller and multibody system models, currently presents a challenge. Although several central product models have been developed for multidisciplinary design, none has yet gained, in contrast to geometry-focused central product models, wide acceptance nor reached the status of an international standard. The unmanageable high number of diverse discipline- and application-specific modeling concepts hinders the development of a standardized holistic central product representation.

This thesis investigates the possibility of establishing an interdisciplinary central product model based on the common modular structure of models from various disciplines. Most models which are edited with current state-of-the-art software applications are composed of modular components in order to support the exchange and reuse of model information. Models from different disciplines therefore share common modeling concepts

for the specification of modular model components. However, there is yet no overarching modeling standard to describe the common characteristics of modular model components from various disciplines.

Object-oriented modeling concepts currently mainly describe software modules called objects. Object-oriented modeling concepts are generic and can be used to represent modular components in general. The Unified Modeling Language (UML) has been since its emergence in 1997 the de facto standard for object-oriented modeling.

This thesis examines the use of the object-oriented modeling concepts of the UML to uniformly describe widely used application-specific geometric, dynamic and multibody system models in a central product model. Application-specific model information was represented in UML through generic UML modeling concepts in combination with lightweight UML extensions in the form of stereotypes. UML profiles regrouped stereotypes which corresponded to a specific modeling application. The automatic translation of UML model information into the specific models and vice versa was implemented in order to test and validate the application-specific UML profiles.

The UML-based central product model was used in several test cases to automatically generate consistent models for the simulation and evaluation of various product configurations. The test cases included models for the simulation of slider-crank mechanisms, the evaluation of cabin pressure control systems, the design of conveyor system configurations, the evaluation of satellite configurations and the generation of customized aircraft geometry. The workflows within the test cases included the automatic creation and modification of UML models as well as the invocation of data exchange connections. The workflows were described in executable UML activity diagrams or Java programs.

The thesis demonstrates that the UML can be used beyond conventional software modeling to establish a central holistic product representation. The modeling concepts of geometric, dynamic and multibody system models were translated mostly according to one-to-one mappings into corresponding UML modeling concepts with their respective stereotype. As a result, the specific model information is easily recognizable in the UML-based central product model. Furthermore, the use of a UML-based central product model is facilitated for the many modelers who are already familiar with the widespread and standardized UML modeling language.

Kurzfassung

Die Dissertation untersucht die Eignung der Unified Modeling Language (UML) für den Aufbau eines zentralen Produktmodells zur multidisziplinären Produktdatenintegration. Im rechnerunterstützten Produktentwurf werden spezialisierte disziplinspezifische Software-Anwendungen zur Modellierung und Simulation von verschiedenen Produktaspekten verwendet. Abhängigkeiten zwischen Modellen treten häufig auf infolge der redundanten Verteilung von Produktdaten über mehrere Modelle und der Beziehungen zwischen verschiedenen Eigenschaften von unterschiedlichen Modellen. Änderungen an einem Modell erfordern eine Aktualisierung von abhängigen Modellen. Konsistenz zwischen Daten aus unterschiedlichen Produktmodellen wird automatisch durch Datenaustausch-Software sichergestellt.

Die Verwendung eines zentralen Produktmodells ermöglicht eine Reduzierung der Anzahl an Datenaustauschschnittstellen. Zentrale Produktmodelle speichern über mehrere Modelle verteilte Produktinformationen und gewährleisten Datenkonsistenz, indem sie mit spezifischen Modellen sternförmig vernetzt sind. Zentrale Produktmodelle sind von besonderem Nutzen in Entwurfsszenarien, welche eine hohe Anzahl an Abhängigkeiten zwischen Modellen sowie Modellanpassungen aufweisen.

Die Integration von geometrischen und davon abgeleiteten Produktmodellen wurde in mehreren zentralen Produktmodellen erfolgreich durchgeführt. Allerdings stellt die Integration von diverseren Produktmodellen, wie zum Beispiel Geometrie-, Software-, Regelung- und Mehrkörpersystem-Modellen, eine Herausforderung dar. Obwohl mehrere zentrale Produktmodelle zur multidisziplinären Produkdatenintegration entworfen worden sind, hat keins bis jetzt, im Gegensatz zu geometriefokussierten zentralen Produktmodellen, entweder eine breite Akzeptanz gefunden oder den Status eines internationalen Standards erreicht. Die unübersichtlich hohe Anzahl an unterschiedlichen disziplin- und anwendungsspezifischen Modellierungskonzepten erschwert das Erstellen einer standardisierten ganzheitlichen zentralen Produktbeschreibung.

Die Dissertation untersucht einen Ansatz zur Erstellung eines interdisziplinären zentralen Produktmodells basierend auf der gemeinsamen modularen Struktur von Modellen aus unterschiedlichen Disziplinen. Die meisten Modelle, die mit modernen Software-

Anwendungen erstellt worden sind, bestehen aus modularen Komponenten, um die Wiederverwendung und den Austausch von Modell-Informationen zu vereinfachen. Modelle aus verschiedenen Disziplinen weisen deshalb gemeinsame Modellierungskonzepte auf, um die Kapselung, Klassifikation und Interaktionen von modularen Modellkomponenten zu beschreiben. Allerdings gibt es noch keinen domänenübergreifenden Modellierungsstandard, um die gemeinsamen Eigenschaften von modularen Modellkomponenten aus unterschiedlichen Disziplinen zu beschreiben.

Objektorientierte Modellierungskonzepte werden bis jetzt hauptsächlich zur Beschreibung von Software-Modulen, die als Objekte bezeichnet werden, verwendet. Sie sind generisch und können modulare Modellkomponenten im allgemeinen repräsentieren. Die UML is seit 1997 ein weitverbreiteter Standard zur objektorientierten Modellierung.

Die Dissertation untersucht die Verwendung der objektorientierten UML-Modellierungskonzepte für eine einheitliche Repräsentation von weitverbreiteten anwendungsspezifischen Geometrie-, Dynamik- und Mehrkörpersystem-Modellen. Anwendungsspezifische Modellinformationen wurden im UML-Modell durch generische UML-Modellierungskonzepte sowie leichtgewichtige UML-Erweiterungen in der Form von Stereotypen repräsentiert. Anwendungsspezifische UML-Stereotypen wurden in UML-Profile zusammen gruppiert. Die automatische bidirektionale Übersetzung zwischen dem zentralen UML-Modell und den spezifischen Modellen wurde zur Überprüfung und Validierung der anwendungsspezifischen UML-Profile implementiert.

Das auf UML basierende zentrale Produktmodell wurde in mehreren Testfällen zur automatischen Generierung von konsistenten Modellen eingesetzt, um die Simulation und Bewertung von unterschiedlichen Produktkonfigurationen zu ermöglichen. Die Arbeitsflüsse innerhalb der Testfälle beinhalteten die automatische Erstellung und Anpassung des zentralen UML-Modells sowie den Aufruf der Datenaustauschschnittstellen. Sie wurden in ausführbaren UML-Aktivitätsdiagrammen oder Java-Programmen beschrieben.

Die Dissertation zeigt, dass die UML über die reine Software-Modellierung hinaus zur zentralen ganzheitlichen Produktrepräsentation verwendet werden kann. Die Modellierungskonzepte von Geometrie-, Dynamik- und Mehrkörpersystem-Modellen liessen sich in den meisten Fällen durch eins-zu-eins Abbildungen in entsprechende UML-Modellierungskonzepte mit Stereotyp übersetzen. Dadurch lassen sich domänenspezifische Modellinformationen im zentralen UML-Produktmodell leicht wiedererkennen. Zusätzlich fällt der Einsatz eines zentralen UML-Produktmodells vielen Modellierern, die mit der weitverbreiteten und standardisierten UML Modellierungssprache schon vertraut sind, leichter.

Chapter 1

Introduction

1.1 Product data consistency

The computer-aided design of multidisciplinary products involves the use of specialized discipline-specific software applications in order to model and simulate various product aspects. Dependencies between models are thereby frequent as the same product information often appears redundantly in various engineering models. In addition, dependencies between models exist due to relationships between distinct features of various models. A change in one model then requires the update of dependent models. The simulation of models based on inconsistent data is meaningless and can lead to subsequent wrong design decisions. The synchronization of models is therefore necessary.

As an example, the dependent models of a slider-crank mechanism are presented in Fig. 1.1. The represented mechanism consists of a slider which can be displaced along its track by applying a torque to the crank. A controller is responsible for computing the torque in order to position the slider according to a specific target. The mechanism is described by a 3D geometric model to specify the decomposition of the mechanism in parts, a multibody system model to simulate the dynamic behavior of the mechanism when a torque is applied to the crank and a controller model to represent the control logic and compute the torque. The inertial properties of the parts such as mass and moments of inertia are present redundantly in both the mechanism's geometric and multibody system models. Furthermore, the multibody system model is embedded in the controller model in order to simulate the controlled motion of the mechanism. Due to these dependencies, a change in the mechanism's geometric model thus requires the subsequent update of the mechanism's multibody system model and a new simulation of the controlled motion to validate the mechanism's controller.

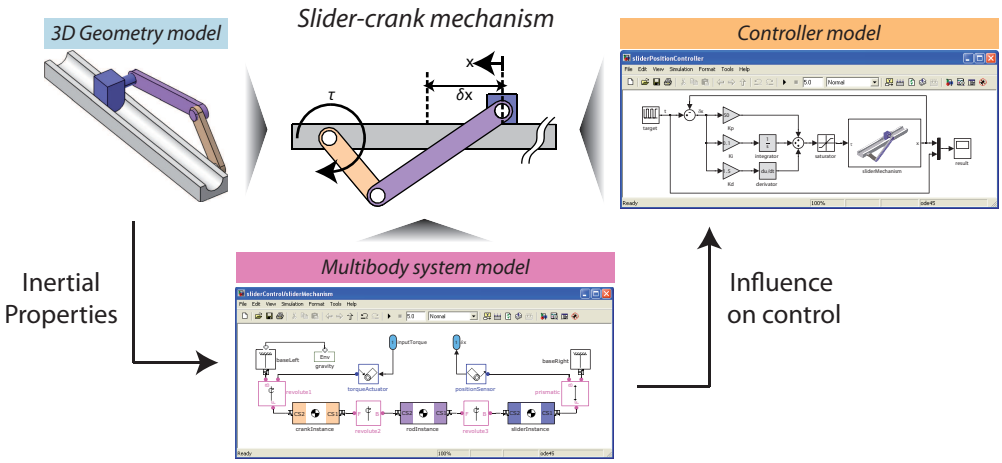

Figure 1.1: Various dependent models of a controllable slider-crank mechanism

The manual update of models by engineers is an unproductive task. Models need to be updated frequently if product requirements often change or if many iterative design modifications are required in order to reach an optimal design configuration. Furthermore, the synchronization of models may require the update of large amounts of data. In the example of the slider-crank mechanism, the update of the multibody system model based on a new geometric model involves the update of the initial position, orientation, center of gravity, mass and moments of inertia of every moving part. Although the slider-crank mechanism only consists of a few parts, the update of the multibody system model according to a new geometric model therefore requires the update of many parameters. The manual update of models by engineers is as a consequence error-prone and time-consuming. Instead, a framework for automatic model updates is needed in order to efficiently guarantee data consistency across various product models.

1.2 Central product models

Data consistency between models is achieved automatically through model-to-model data exchange software. The development and maintenance of each specific data exchange connection represents a large effort. The use of a central product model enables to reduce the required number of data exchange connections. A central product model stores the redundant product information which is spread across several models and achieves data consistency between the specific models through data exchange connections between itself and the specific models as in a hub-and-spoke network. As shown in Figure 1.2, the bidirectional linking of n specific models via a central product model requires only $2n$ connections while the equivalent direct linking of models needs $n(n-1)$ connections. The

scenario with a central product model is therefore preferable to the direct data exchange between specific models when *n* is greater than 3 as it involves fewer data exchange connections to achieve data synchronization.

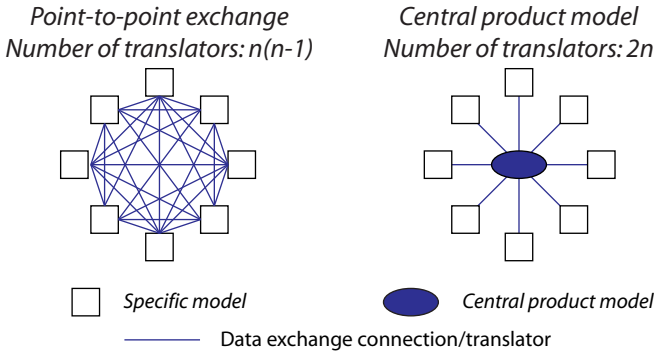

Figure 1.2: Inter-model data exchange: point-to-point or via a central product model

In addition, a central product model can represent information from different specific models and as a consequence describe inter-model dependencies. A central product model can thus provide through its integrative role a holistic product overview of various multidisciplinary dependencies. This is especially useful in large design projects in which it is hard to keep track of all product subsystems and dependencies.

Furthermore, a central product model can be used to facilitate product customization. Product aspects which are described in the central product model can be automatically translated into specific models. It is thus more efficient to change product aspects once in the central product model and subsequently automatically generate conforming specific models than to individually customize several specific models.

The central product model can for example serve as a common product parameter repository for the automatic update of specific model parameters. Due to inter-model dependencies, parameter modifications in one model have to be forwarded to other models. The propagation of parameter modifications between models is directed since the mapping of parameters between models is most often not bijective. As an example, the moment of inertia parameter of a multibody system model may be computed from sizing parameters of a geometric model but not vice versa as an infinite number of possible sizing configurations may satisfy a specific moment of inertia. In general, a mapping between vector spaces $\mathbb{R}^i$ and $\mathbb{R}^j$ is only bijective if their dimensions are equal ($i = j$). However, parameter sets of different product models usually form vector spaces with different dimensions ($i \neq j$). As a result, mapping functions between parameters of distinct product models are most often not bijective and parameters modifications in one model are thus forwarded to other models according to a directed information flow from a higher

dimensional into a lower dimensional design representation ($i \geq j$). The central product model can contain all relevant product parameters and form the design representation with the highest dimensionality from which parameters in design representations with lower dimensionality can be updated by means of a projection as in Rudolph [141].

Most central product models are used for the design of specific types of products. A prominent example is the Standard for the Exchange of Product Model Data (STEP) [64] part AP214 [66] for automobile design. Another important example is the Industry Foundation Classes (IFC) standard [22] for the design of buildings. Within the aerospace industry, a European-funded project for example developed a product model for the multidisciplinary design and optimization of blended wing-body configurations [100]. This central product model contained the full parametric description of the aircraft and was linked with software tools for aerodynamic, structural, dynamic and flight mechanics analysis [94].

Product models devoted to a single product category can more easily include detailed product information but cannot be used for the design of other product types. The invested effort in the implementation and maintenance of the data exchange software between the specific models and the central product model is hence limited to the design of specific product types. However, some central product models are generic enough to be employed for the design of various product types. This is more advantageous since the same central product model and associated data exchange software can be reused for the design of products across a wide range of industry sectors. The existing central product models vary according to the type of product information they can represent and in their support for data exchange connections with state-of-the-art modeling applications.

Among the commercial solutions, Product Lifecycle Management (PLM) and Product Data Management (PDM) systems include features for product modeling and the management of multidisciplinary dependencies. A standardized approach is the STEP PDM schema [161] for a common formal representation of product information in PDM systems. But PDM data models such as the STEP PDM schema do not intend to incorporate the fine granularity of detailed models of various disciplines but instead mainly concentrate on the management of geometric parts and documents [153]. As a consequence, the poor support for the integration of mechanic, electronic and software components is the main weakness of existing PDM/PLM solutions [1].

Many product models have been developed in academia and in the industry to integrate product information from several domains and enable their interdisciplinary exchange. NIST's Core Product Model (CPM) [155] is for example focused on product lifecycle management and captures at an abstract level product features, artifacts, behaviors, spec-

ifications and design rationale. Boeing's Integrated Product Design Environment (IPDE) for example formed a central product model to support multidisciplinary product development by integrating CAD, CFD, FEA and manufacturability features [98]. Another example is the collaborative design system called Constraint Linking Bridge (Colibri) [86] which is used to solve dependencies between the parameters of mechatronic models such as geometric, controller and hydraulic models.

Although several central product models have been developed for the multidisciplinary design of diverse products, no central product model has yet gained wide acceptance nor reached the status of an international standard. The unmanageable high number of diverse discipline- and application-specific modeling concepts hinders the development of a holistic central product representation. The adoption of a central product model and its modeling concepts depends on the simplicity with which modelers can describe detailed dependencies between various product models. An example of detailed dependencies between models is the required synchronization of the inertial properties of the slider-crank mechanism which are situated at a detailed level within the respective geometric and multibody system models (Fig. 1.3).

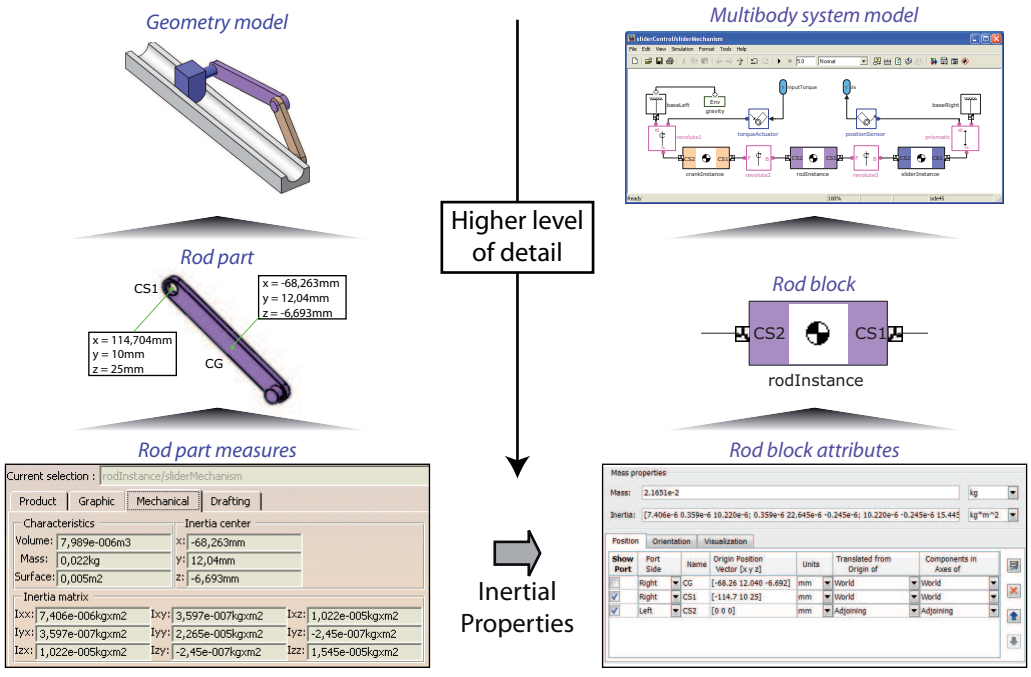

Figure 1.3: Dependencies between models at a deep nested level

The central product model cannot represent various product aspects with the same modeling concepts as specialized discipline- and application-specific models due to the rich diversity of modeling concepts in the various engineering disciplines. Technical product information is described in models which highly differ in terminology and represen-

tation. A product's three-dimensional geometric model is for example different than a product's two-dimensional block diagram of a multibody system model. In addition, product models within the same engineering discipline can be dissimilar due to distinct modeling paradigms. The geometric model of a three dimensional product can for example be defined either based on a boundary representation or a constructive solid geometry representation or both. Furthermore, models can differ due to distinct application-specific modeling concepts. The precise mathematical definition of a spline can for example vary from one geometric modeling application to another. As a result, product information, which is related to the computer-aided design of a multidisciplinary product, is scattered over a wide range of heterogeneous discipline- and application-specific models. Central product models can therefore not include all the different modeling concepts of various discipline- and application-specific models.

Instead, central product models are comprised of general modeling concepts which correspond to commonly used specific modeling concepts. Central product models can thus represent numerous specific modeling concepts through a manageable set of generic modeling concepts. Figure 1.4 shows exemplarily the one-to-one mapping between respective specific and generic modeling concepts. The central product model can for example include a modeling concept called *module* in order to represent modular components from different disciplines such as *parts* of geometric models or *blocks* of multibody system models. Similarly, the central product model can include generic modeling concepts named *model* and *property* to refer respectively to specific models and properties. The generic modeling concepts can further be detailed through standardized extension mechanisms in order to have the same semantics as specific modeling concepts. The extension of generic modeling concepts is for example widely used in model-based software engineering [112] and recently also in model-based systems engineering [80].

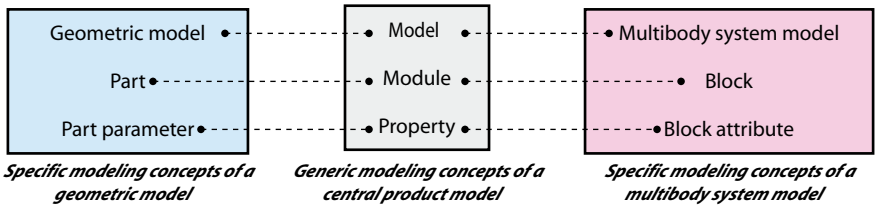

Figure 1.4: Mapping between specific and generic modeling concepts

Dependencies between specific models can be described in the central product model as dependencies between equivalent model representations. For instance, the dependency between the rod part measure of the geometric model and the rod block property of the multibody system model (Fig. 1.3) can be represented as a dependency between the respective generic properties within the central product model (Fig. 1.5).

Dependencies can involve information which is situated deep within the hierarchical structure of models. Deep nested model information is easily identified based on its location within the model hierarchy. The identification of the rod part mass within the geometric model in Figure 1.3 for example requires the reference to its containing rod part and geometric model and can be described through the following scheme: Geometric model/Rod part/Mass measure. In order to easily identify the equivalent deep nested specific model information in the central product model, the model information needs to be represented in the central product model along with its hierarchical structure. The same identification scheme based on the model hierarchy can then be used to identify information in specific models and in the central product model. This facilitates the identification of deep nested model information in the central product model. In the previous example, the rod part mass of the geometric model is therefore represented in the central product model along with its owning rod part and geometric model. In other words, the deep nested model information involved in dependencies is represented in the central product model along with its modeling context.

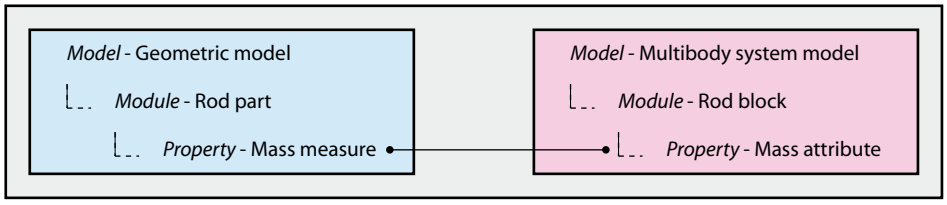

Figure 1.5: Representation of specific model information and deep nested inter-model dependency within the central product model

The choice of generic modeling concepts of the central product model is critical for the capability of the central product model to represent product information from a wide range of disciplines and modeling applications and thus to represent detailed inter-model dependencies. Discipline-specific engineers can only easily describe their specific product information within the central product model if the modeling concepts of the specific models can be mapped one-to-one to the generic modeling concepts of the central product model. As an example, the limited generic modeling concepts of the central product model in Fig. 1.4 are related to static artifacts and cannot easily represent flows of data such as in the Simulink-specific dynamic system model of Fig. 1.1. The central product model would for example need to include an additional *information flow* modeling concept in order to represent specific flows of data. The central product model thus needs to include modeling concepts which are commonly used in models throughout various disciplines.

The integration of geometry and therefrom derived models such as structural analysis or computational fluid dynamics models has already been successfully addressed in numerous central product models. However, the integration of models for the design of mechatronic products, such as geometric, controller and multibody system models, currently presents a challenge due to their diversity [91]. No central product model is for example known to solve the interoperability problems of the slider-crank mechanism scenario as shown in Fig. 1.1 and Fig. 1.3 as it involves multidisciplinary models and the exchange of deep nested application-specific information. However, the integration of multidisciplinary models in a central product model is of utmost importance for the design of mechatronic products ranging from aerospace engineering to modern manufacturing facilities.

The modeling concepts of the current central product models are either too discipline-specific or too abstract. As a result, current central product models are either limited in their capacity to represent information from a wide range of disciplines or in their capacity to capture deep nested application-specific model information. Discipline-specific modeling concepts can easily represent detailed information such as specific deep nested model information. However, they are restricted to specific disciplines and are therefore not suitable within a central product model for multidisciplinary product data integration. On the other hand, abstract modeling concepts can represent specific modeling concepts of various disciplines. However, the current central product models do not include enough abstract modeling concepts to enable a simple one-to-one correspondence with modeling concepts of various state-of-the-art application-specific models. Without a simple one-to-one mapping between generic and specific modeling concepts, modelers do not easily recognize their specific model information within the central product model and inter-model dependencies are as a consequence hard to describe. The resulting loss of time in using an incomprehensible central product model counterbalances its benefits and reduces its usefulness.

As research in multidisciplinary product data integration has only been undertaken recently, there is yet no official widely used term to refer to a “central product model”. The term “integrated product model” is also often used synonymously. The neutral term “central product model” is used throughout this thesis because it does not emphasize a proprietary nor a specific type of model such as the Multidisciplinary Collaborative Design Product Model [97] or the Core Product Model [97].

1.3 UML-based product data integration

Although models from different engineering disciplines are highly diverse, most models which are edited with current state-of-the-art software applications share common modeling concepts in order to support modular design. The capacity to easily exchange and reuse model components across several models promotes flexibility and productivity in modeling. This avoids the time-consuming creation of models from scratch. Most models therefore share common modeling principles in order to describe modular model components. Common characteristics of modules include their in- and outputs, their hidden and visible outward-facing information, as well as templates and instances.

A central product model should include modeling concepts to describe modules and their interactions as they are common to many state-of-the-art models from various disciplines. Engineers can then easily recognize their modular-structured model information within the larger central product model. Current central product models do not include, or only partly include modular modeling concepts which are increasingly used across various engineering disciplines.

There is currently no widely accepted standard to represent with general overarching concepts modules from different disciplines. However, by comparing the various engineering disciplines, it is noticeable that modularity is especially important in software engineering. Software design is an engineering discipline in which changes and updates are more frequent than in other engineering disciplines as software code is simpler to modify than mechanical engineering components which require manufacturing and resources. As a result, concepts to promote modular design are more frequently used in software engineering than in other engineering disciplines. Sophisticated programming concepts have been developed in software engineering to support modularity, whereby the most prominent are the object-oriented programming concepts. They consist of encapsulating variables and functions into modular units called objects. Graphical models of object-oriented software represent the classification, communication and internal structure of software objects.

Although object-oriented modeling is currently mainly used for software modeling, object-oriented modeling concepts are generic and can be used to describe various modular structures. As the term object already suggests, an object can represent a software object as well as a physical component or a model component. Object-oriented modeling is thus not restricted to software modeling. As object-oriented modeling concepts include modeling concepts amongst others to describe the composition, encapsulation and templates of objects, the existing object-oriented modeling concepts can be reused in the context of a central product model to describe modules from different engineering dis-

ciplines. Both the central product model and the discipline-specific models would thus share a common modular structure. Engineers from various disciplines could then more easily recognize their discipline-specific model information within the central product model.

As a central product model is to be used across several disciplines, it addresses many parties and therefore requires standardization. Each type of central product model requires special training and dedicated conversion tools. A standardized central product model would thus be desirable as it would eliminate the confusion caused by different central product models. A standardized central product model would also reduce the development costs of data exchange software and thus contribute to higher interoperability.

The Unified Modeling Language (UML) has been since its emergence in 1997 the de facto standard for object-oriented modeling and is widely used in software engineering. A standardized central product model could therefore be built upon the already standardized object-oriented modeling concepts of the UML. The redefinition of semantically similar object-oriented modeling concepts would instead most probably lead over time to confusion among engineers. Furthermore, the standardization process involved in the definition of new object-oriented modeling concepts specifically for a central product model would likely, as any standardization process, be a lengthy process which would not necessarily end up in a consensus among experts. Furthermore, as UML is already a widely adopted standard in software engineering, the existing large software and documentation support for UML modeling would facilitate the introduction of a central UML-based product model in an industrial context.

Thimm et al. [158] introduced the potential of modeling a product's lifecycle by using the UML and presented the UML as the most promising candidate to use as a unique language for all product lifecycle management stages as it offers an information-rich representation which can be translated into other representations. Johansson and Detterfelt [79] identified the UML as an interesting approach for the modeling of multidomain system products due to its easy understandability by engineers with significantly different backgrounds. However, the UML has not yet been investigated in view of establishing a central product model which can represent state-of-the-art model information from typical mechatronic disciplines.

This thesis investigates the capability of the UML to describe application-specific model information from various disciplines. The integration of geometric, controller and multibody system models is required in many mechatronic products which abound in aerospace, automobile or manufacturing products. The approach of reusing the UML to support the representation of various models in a central product model is examined

by representing state-of-the-art application-specific geometric, controller and multibody system models in a common UML-based central product model.

CATIA[1], Simulink[2] and SimMechanics[3] are state-of-the-art software applications for the respective authoring of geometry, controller and multibody system models. These applications were chosen among others to prove the integration capabilities of a UML-based central product model. Further application-specific model information was also integrated into UML, including SolidWorks[4] and VRML[5] models for geometry, Excel[6] for spreadsheets and Matlab[7] for constraint processing. Figure 1.6 summarizes the domains and the applications which have been integrated within the UML-based product model. The necessary UML lightweight extensions, in other words UML profiles, for a mapping of the respective application-specific engineering models into a common UML-based central product model are presented in the thesis.

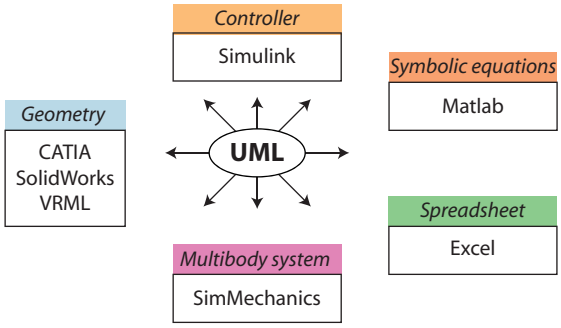

Figure 1.6: Integrated applications within the UML-based central product model

The mappings between the application-specific models and the UML-based central product model were to a large extent bijective and thus easy to understand. This shows that the UML already consists of a wide range of generic modeling concepts which can cover a variety of specific modeling concepts from various disciplines. The UML has thus the capacity to model not only software but also product information from various disciplines involved in mechatronics including geometry, dynamic systems and multibody systems. The thesis thus demonstrates that the UML can be used beyond conventional software modeling in order to establish a standard central product model due to its object-oriented modeling principles, openness and extensibility.

[1] DASSAULT SYSTEMES, http://www.3ds.com/products/catia/
[2] The MathWorks, Simulink, http://www.mathworks.com/products/simulink/
[3] The MathWorks, SimMechanics, http://www.mathworks.com/products/simmechanics/
[4] SolidWorks 3D CAD Design Software, http://www.solidworks.com/
[5] VRML Virtual Reality Modeling Language, http://www.w3.org/MarkUp/VRML/
[6] Excel, http://office.microsoft.com/en-us/excel/
[7] The MathWorks, Matlab, http://www.mathworks.com/products/matlab/

The UML-based central product model was used in several projects in partnership with academia and industry. The test cases included the evaluation of cabin pressure control systems, the automated design of conveyor system configurations, the automated evaluation of satellite configurations and the generation of customized geometric aircraft models. The test cases enabled to validate the capabilities of the UML-based central product model to represent model information from various mechatronic disciplines (geometry, multibody dynamics, control), specify inter-model dependencies and automatically translate information within the central product model into the specific models and vice versa. Consequently, the UML-based central product model was used in all test cases to generate various consistent sets of specific models which corresponded to various product configurations. The UML-based central product model can thus be used, in combination with specialized discipline-specific models, for the consistent design of customized multidisciplinary products.

1.4 Outline

The most common models for product data integration are presented in Chapter 2. The prominent STEP standard is presented as well as some STEP-based product data integration approaches. Relevant industry sector standards for the building and process plant industries are reviewed. Ontologies are also described as they have contributed to product data integration in various disciplines.

Chapter 3 presents the motivation to establish a central product model based on UML. The necessity to capture information from a wide range of disciplines through generic modeling entities is shown. Furthermore, the common decomposition of application-specific models into modular components is presented as well as the generic object-oriented modeling concepts which can describe modular components from different disciplines. The UML as de facto object-oriented modeling standard is introduced and is compared to other standardized generic modeling languages in view of establishing a central product model. The main UML modeling entities and the lightweight extension mechanism in the form of stereotypes are demonstrated as well as the UML-based product data integration approaches.

Chapters 4 to 6 describe the mapping of different application-specific models into UML. The approach is demonstrated in detail for the design of a slider-crank mechanism, which is simple to understand, but whose application-specific models cover a large scope of application-specific modeling concepts. Chapter 4 presents the UML extensions to describe geometric models. The approach is shown with CATIA and SolidWorks, which

are two state-of-the-art 3D geometry authoring tools, as well as with VRML, which is a widely used format capable of representing static and animated 3D objects in freely available viewers.

Chapter 5 shows the translation of dynamic system models into UML. Simulink is a software application used to model and simulate dynamic systems in general while SimMechanics is specialized in the modeling and simulation of multibody systems. Both applications use block diagrams. However, the modeling concepts in the block diagrams are not identical. As a result, their mapping into UML occurs differently. Chapter 5 presents the mapping of Simulink models into UML activity diagrams and SimMechanics models into UML composite structure diagrams.

Chapter 6 describes the UML-based representation of product data originating from Excel spreadsheets. In order to achieve data consistency between the values from different applications, symbolic equations within the central UML-based product model need to be solved. Chapter 6 presents the UML-based description of symbolic equations and their resolution through mathematical toolboxes. Complex computations need to refer to built-in or user defined mathematical functions. The UML representation of Matlab-specific functions is presented as well as their processing.

Chapter 7 provides a review of the possibilities to establish links within the UML-based central product model in order to guarantee data consistency between UML-based representations of different application-specific models. Chapter 7 also exhibits the execution of Java programs and UML activity diagrams in order to use the UML-based product model in an automated design workflow. The frameworks that have been used to implement the translators between the UML-based central product model and the application-specific models are described.

Chapter 8 provides a review of test cases which highlight the consistent design of customized application-specific models by means of a UML-based central product model. The case studies include models for the evaluation of cabin pressure control systems, the automated design of customized conveyor systems, the automated evaluation of different satellite configurations and the customizable generation of aircraft geometry.

Chapter 9 summarizes the results and presents an outlook.

Chapter 2

Models for product data integration

The sharing of product data on a large scale, such as in an international context between participants from different companies and disciplines, is achieved through standardization of product models. Standards for specific disciplines or industry sectors have therefore emerged. Section 2.1 presents the integration capabilities of the STandard for the Exchange of Product Data (STEP) which is the most prominent standard in mechanical engineering. Integration approaches based on STEP are also reviewed. In addition, standards outside the scope of STEP have been developed for specific industry sectors. Section 2.2 describes the Industry Foundation Classes (IFC) and ISO 15926 which are respectively international standards for the building and process plant industries. Furthermore, ontologies have been used in engineering design to formally describe product-related knowledge as they are composed of simple subject-predicate-object expressions which can easily describe relations between different heterogeneous product data sources. The domain-independent ontology modeling languages are therefore employed for the integration of product information from different systems or disciplines. Ontologies and ontology-based integration approaches are shown in Section 2.3.

2.1 STEP

Around 1980 several national standards, such as the “Initial Graphics Exchange Specification (IGES)” [162] from the United States, “Standard d’Echange et de Transfert (SET)” [6] from France and “Verband der Automobilindustrie - Flächenschnittstelle (VDA-FS)” [163] from Germany were defined to enable the exchange of product geometry between different Computer-Aided Design (CAD) systems. To avoid incompatibility between the national standards, a multinational initiative was started in 1984 to develop a single international standard. In 1994, the multinational effort resulted in the ISO 10303

standard with the official title “Industrial automation systems and integration - Product data representation and exchange”, known as the STandard for the Exchange of Product Model Data (STEP). Section 2.1.1 presents the family of STEP standards and their underlying data modeling language EXPRESS. Section 2.1.2 shows the different integration approaches based on the family of STEP standards.

2.1.1 Overview

Originally, the standard was to offer an Integrated Product Information Model (IPIM) capable of capturing product data from several disciplines [82]. However, this undertaking was too time- and resource-consuming. Rather, the development of the STEP standard was driven by participants who had received funding to work on some particular aspects of product data. Eventually, STEP did not become a single standard covering several disciplines, as the name suggests, but a collection of standards for single disciplines which are called Application Protocols (APs).

The development of a STEP AP consists of four major stages [51, 4]. The first describes the usage scenario of a STEP AP through an Application Actitity Model (AAM). In a second stage, the requirements for the STEP AP are derived from the AAM and specified in an Application Reference Model (ARM). The standardized data structure to capture information from a specific discipline is described in a third stage in an Application Interpreted Model (AIM) based on the requirements of the ARM. In other words, the AIM of a STEP AP specifies a schema according to which STEP data is to be structured in files. In a fourth stage, code is implemented based on this schema to enable the standardized exchange of product data.

The initial release of STEP in 1994 included two STEP APs which were AP201 for “Explicit Draughting” and AP203 for “Configuration controlled 3D design of mechanical parts and assemblies”. Currently, 18 STEP APs, listed in Table 2.1, have reached the status of international standard and are used typically in the aerospace, automotive and shipbuilding industries. However, not all STEP standards are equally well adopted and supported by software vendors. At present, only parts of STEP AP203 and AP214, respectively for product geometry and automotive product data, are implemented by major CAD systems [151].

Each STEP AP specifies a schema in an Application Interpreted Model to define the data structure of related STEP files. The schemas of STEP APs are defined using the EXPRESS data modeling language [63] which is also a standard within ISO 10303. EXPRESS was developed in the 1980s because the existing data modeling languages of the time such as the extended entity-relationship model (EER) had been conceived to repre-

STEP APs	Engineering Domain
AP201	Explicit draughting
AP202	Associative draughting
AP203	Configuration controlled 3D designs of mechanical parts and assemblies
AP204	Mechanical design using boundary representation
AP207	Sheet metal die planning and design
AP209	Composite and metallic structural analysis and related design
AP210	Electronic assembly, interconnect, and packaging design
AP212	Electrotechnical design and installation
AP214	Core data for automotive mechanical design processes
AP215	Ship arrangement
AP216	Ship moulded forms
AP218	Ship structures
AP224	Mechanical product definition for process planning using machining features
AP225	Building elements using explicit shape representation
AP227	Plant spatial configuration
AP232	Technical data packaging core information and exchange
AP239	Product life cycle support
AP240	Process plans for machined products

Table 2.1: STEP APs having reached the status of International Standard (as of Sep 2009)

sent business information and not product data. EXPRESS included new concepts such as multiple inheritance and composition rules for the formal representation of product data [39]. An EXPRESS model to define drawings composed of points and lines is represented in Fig. 2.1 after an example in Peak et al. [131]. The point and line entities for example have an inheritance relationship with the shape entity which is specified through the keyword "SUBTYPE OF". As a result, the point and line entities inherit the label attribute of the shape entity. An EXPRESS schema can also be described graphically according to the EXPRESS-G notation. However, EXPRESS-G can only describe a subset of EXPRESS.

A STEP file is displayed in Fig. 2.1 left. The header of the file specifies the corresponding file schema and can include additional metadata such as file name and author. The file describes a drawing composed of two lines and three points. Each entity instance has a number as identifier so that it can be referenced by other entity instances. The attribute values of an instance are listed next to the instance type. The data is written in STEP files in an ASCII-based syntax according to Part 21 of ISO 10303 [65]. Each STEP file or EXPRESS model instance is therefore also called a Part 21 file.

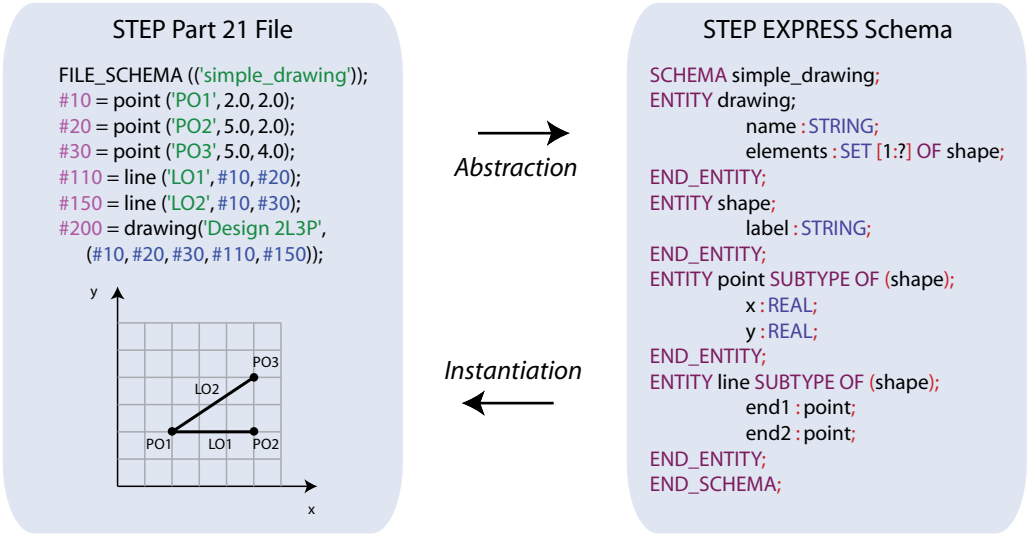

Figure 2.1: STEP Part 21 file and related EXPRESS schema (after Peak et al. [131])

All STEP APs are defined based on common schemas from the Integrated Generic Resources. These schemas for example describe the “fundamentals of product description and support” (ISO10303 part 41 [71]) or the “geometric and topological representation” (ISO10303 part 42 [72]).

Within the scope of STEP, an Application Programming Interface (API) to access STEP P21 files was defined as part 22 of ISO10303 [68] and named Standard Data Access Interface (SDAI). Several bindings exist to support the use of SDAI in programming languages such as C++, C and Java. Each EXPRESS-defined STEP schema is associated with a specific SDAI derived from the EXPRESS schema.

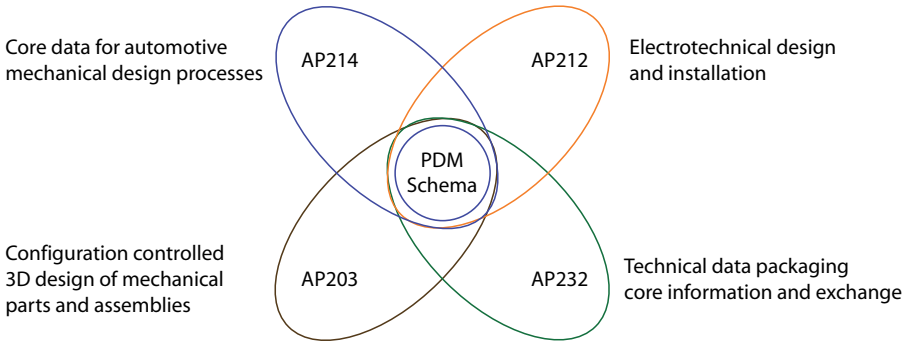

Figure 2.2: STEP PDM schema common to four STEP APs (after Srinivasan [152])

The STEP PDM schema [161] is of particular importance as it is common to several STEP domain-specific models. Independent of the domain, the management of documents is important in a design environment in which many engineers work on the same files. Product Data Management (PDM) systems typically manage the traditional meta-

data including document author, approver, version, history, and also geometric information such as product assembly structures, part numbers and bills of material. The STEP PDM schema is the result of ISO-based standardization efforts regarding PDM information. As the management of documents is common to several domains, the STEP PDM schema is a common subset of several STEP APs such as the AP203 for "Configuration controlled 3D design of mechanical parts and assemblies", the AP212 for "Electrotechnical design and installation", the AP214 for "Core data for automotive mechanical design processes" and the AP232 for "Technical data packaging core information and exchange" [152] (Fig. 2.2).

The effort required of a software vendor to support a complete STEP AP, in other words to implement a translation between a software vendor format and a neutral STEP AP format, might be too costly. STEP APs have therefore been decomposed in specific subsets called Conformance Classes (CC) in order to facilitate the adoption by software vendors of only certain parts of STEP APs. As a consequence, STEP currently enables the exchange of subsets of domain-specific product data between applications.

The original vision of STEP aiming at replacing the direct translators between applications by a neutral format covering several disciplines has yet only partially materialized. This goal can only be reached if a large group of end-users explicitly demand standard-compliant applications and thus influence the standards adoption policy of software vendors. However, the process of standardization and industry adoption is too slow for many end-users who are forced to rely on the error-free exchange of data by "de facto" proprietary standards set by software vendors covering many disciplines and having a large market share [49]. Nevertheless, about 1000 person-years of effort have been required so far to create the family of STEP standards [131] which together make STEP the biggest standard within ISO. STEP is the most prominent standard for product data exchange and companies realize important savings through its use. According to an estimation in 2002, the reduction of interoperability costs through the discipline-specific STEP standards resulted in savings amounting to $150 million per year [46].

2.1.2 Integration approaches

The STEP APs enable the neutral exchange of product data in specific domains as shown in Table 2.1. There is no limitation about the scope of a STEP AP. The STEP AP214 for "Core data for automotive mechanical design processes" for example integrates geometric data, PDM data as shown in Fig. 2.2 and information specific to automobiles. Theoretically, several STEP APs can be regrouped into a single STEP AP. Many use case scenarios require a neutral exchange of product data across several domains. However,

most STEP-based integration approaches covering several domains do not end up as an official STEP standard as the standardization process is very slow.

As the ISO10303 STEP standard is composed of many single modular parts, STEP-based integration initiatives can reuse single STEP parts. The STEP standard which is most often used to define a schema and a related SDAI is the EXPRESS modeling language. The European-funded project entitled "A computational design engine incorporating multidisciplinary design and optimization for blended wing-body configuration" for example used the EXPRESS language in 2004 to define a data model [100] for the exchange of product data. A database was implemented based on the EXPRESS-defined data model and the new product data format enabled to share data between several disciplines including geometry, structural and aerodynamic analyses. Generic considerations to extend the approach to other product types were avoided to enable a fast implementation of the integration software. The approach is therefore limited to the design of blended wing-body aircraft.

An EXPRESS schema can be defined by reusing elements of predefined EXPRESS schemas as found in the Integrated Resources or in domain-specific APs. Gu and Chan for example reused the generic STEP schemas in 1995, when few STEP APs had reached a mature status, to create a schema for the integration of a variety of manufacturing domains [57]. As a result, product information related to product geometry, representation, tolerance and assembly was saved in the Generic Product Modeling (GPM) system based on an EXPRESS schema. Interfaces were developed between the GPM system and applications such as AutoCAD and AutoSolid. The GPM approach to connect systems in a computer-integrated manufacturing environment was demonstrated with data related to an engine block. Similarly, Boeing's Integrated Product Design Environment (IPDE) merged in 1999 the STEP APs 203, 209, 214 and 224 for the synthesis of an integrated data schema [98]. The integration approach was based on STEP APs and included extensions for more domains such as aerodynamic analysis, parametric geometry and constraints.

The merging of several STEP-based schemas in one schema to enable the integration of data related to geometry, manufacturing, engineering analyses and document versioning in a product model was very often undertaken. Most approaches are either based on STEP AP214, such as in Chin et al. [26], or on STEP AP203 such as in Song et al. [148]. The Generic Product Modeling Framework (GPMF) of Zhou et al. [172] for example consists of eleven defined EXPRESS schemas based on STEP resources and the STEP AP203. A similar product data representation based on STEP resources and the STEP AP203 was used in the expert system of Zha and Hu [171] for an Integrated Knowledge-based Assembly Planning System (IKAPS).

The design of mechatronic products requires the exchange of product data related to several disciplines such as electronics, mechanics, hydraulics and control. A German initiative chaired by the ProSTEP association[1] developed in 2000 the MechaSTEP neutral data format [128] for the exchange of mechatronics-related product data. MechaSTEP was defined in EXPRESS and according to STEP standardization procedures. The neutral format was validated by implementing interfaces to software applications for multibody system modeling and to the VHDL-AMS[2] standard.

The Architecture, Engineering and Construction (AEC) industry uses several advanced 3D applications for design, analysis and fabrication. Various building product models facilitate the data exchange between diverse applications. The CIMsteel Integration Standard Version2 (CIS/2) [62] is an industry-developed product model widely adopted within the steel construction industry [38]. It was first released in 1995. CIMsteel stands for the Computer Integrated Manufacturing of Constructional Steelwork. The schema of CIS/2 is defined in EXPRESS according to the ISO-STEP technology. Another important building model whose data model was also defined in EXPRESS is the computer-based Integrated Building Design System (IBDS). It was initiated in 1990 within the COmputer Models for the Building INdustry in Europe (COMBINE) project. In contrast to CIS/2 and COMBINE, the ISO 10303 AP225 standard [69] for the description of building elements using explicit shape representation completely relies on STEP technologies. AP225 was released in 1999. Parallel to the development of AP225, an industry consortium formed the Industry Alliance for Interoperability in 1994 due to the slow standardization development within ISO 10303. The alliance released an AEC product model called the Industry Foundation Class (IFC) in 1997 [22] whereby its data models were defined in EXPRESS and implementations took advantage of the EXPRESS tools.

Most STEP-based integration approaches use EXPRESS as data modeling language. The definition of a new EXPRESS data model is thereby often based upon existing EXPRESS schemas. STEP is a very widespread standard for the exchange of product data in the context of typical products relying heavily on geometric models. However, the products of the next generation will be smarter and incorporate more software and electronics. An integration framework spanning more disciplines than geometry and therefrom derived models is therefore necessary.

[1] ProSTEP iViP Association, `http://www.prostep.org`

[2] Very High Speed Integrated Circuit Hardware Description Language-Analog/Mixed-Signal (VHDL-AMS)

2.2 Industry sector-specific standards

The building and the process plant industries, including oil and gas production facilities, display a high degree of multidisciplinarity. Standards have been developed specifically for these domains. Section 2.2.1 presents the Industry Foundation Classes (IFC) as an important standard in the building industry and Section 2.2.2 reviews the ISO 19562 for process plants.

2.2.1 IAI/IFC

The building and facility management industry (AEC/FM) is a typically multidisciplinary sector. Several neutral standards for the exchange of building data have been developed. Next to the CIS/2 [62] and STEP AP225 [69] as described in Section 2.1.1, the Industry Foundation Classes (IFC) [22] is also a prominent standard within the building industry. The IFC specification was developed and maintained by buildingSMART International[3], formerly known as International Alliance for Interoperability (IAI). It was first released in 1997. It has been available since 2006 as release IFC2x3 and as ISO Publicly Available Specification ISO PAS 16739. The IFC data schema represents information entities concerning among others building elements, spaces, properties and shapes which are shared by several software applications in construction or facility management projects. The IFC standard is thereby typically used to bridge different applications related to CAD, CFD, Computer-Aided Facility Management (CAFM), structural and thermal analysis applications, Computer-Aided Architectural Design (CAAD), Heating, Ventilating and Air Conditioning (HVAC) and Quantity Takeoff (QTO) for cost estimation [81, 110, 96, 13].

Although the IFC standard has been used in many projects, the construction sector has difficulty in achieving a single agreed data model [61, 139]. The design of a building encompasses so many domains and aspects that the consensus for a universal building standard seems unlikely. In view of all the imaginable manifestations of a design or building, the convergence towards a unified standard for all possible human interpretations of a building seems impossible and not worth pursuing [37]. Furthermore, the development of a standard requires a considerable degree of upfront work, including achieving consensus, implementing the data model and adapting the applications to the standard. These tasks are especially arduous for a standard attempting to reach an international status. As a result, the standard is not updated at the pace the rapidly evolving business needs. This can turn an initially helpful standard into a hindrance [17]. In addition, the standardization efforts are inefficiently repeated, creating a proliferation of standards. The volume of stan-

[3]IAI Tech International, `http://www.iai-tech.org/`

dards to choose from is often a source of confusion for users [135]. Instead of pursuing an all-in-one approach through a unified international standard, ontologies, as described in the next Section 2.3, are envisioned as a flexible method to integrate heterogeneous data sources related to building design.

2.2.2 ISO 15926

ISO 15926 is a standard for the representation of information related to process plants, including oil and gas production facilities [73]. The scope of the data model covers the entire lifecycle of a facility and its components such as pipes, pumps and their parts. Similar standards for process plants are AP227 [70] and AP221 [67] within the STEP family. AP227 is focused on the plant spatial configuration while AP221 and ISO 15926 also cover functional data. AP221 is better adapted to cover schematic drawings while ISO 15926 can better describe the evolution of a process plant through time [95]. ISO 15926 is the result of several years of effort in developing a standard for process plants. In 1991 a European project called ESPRIT was launched for this cause. Based on this, an industry consortium called the European Process Industries STEP Technical Liaison Executive (EPISTLE) then issued the AP221 standard, which was then adapted to form ISO 15926 in 2003. The construction of buildings, production facilities and equipment is, however, outside the scope of ISO15926.

ISO 15928 is currently mainly used in the oil and gas industry for integrating data across disciplines and business domains. It is for example used for standardized production[4] and drilling reports[5]. As a facility may consist of a multitude of different entities, the ISO 15926 standard also includes generic concepts such as classes, individuals and properties. These non process plant-specific concepts enable to describe entities which are not covered by the casual domain-specific concepts of the standard such as a pipe or a heat exchanger. The ISO 15928 standard can therefore be used theoretically beyond the process plant domain. Although the scope of ISO 15926 is large, literature on projects related to ISO 15926 is scarce. Projects on integrating application-specific data about a process plant lifecycle have for example not been reported [35].

2.3 Ontologies

The word “ontology” is derived from the Greek words “ontos” for “being” and “logos” which in fact has several meanings, among others “logic” or “science”. Ontology is orig-

[4]POSC CAESAR, `http://production.posccaesar.org/`
[5]POSC CAESAR, `http://drilling.posccaesar.org/`

inally a philosophical discipline which studies the existence of entities and their possible categorization within a hierarchy based on similarities and differences. The discipline was inaugurated by Aristoteles in his philosophical work known as Metaphysics. In the early 1990s the term ontology was reused within the artificial intelligence community to define "the basic terms and relations comprising the vocabulary of a topic area" [108], in other words "a formal specification of a conceptualization" [55]. Ontologies thereby describe a domain through concepts, individuals and relations.

Ontologies are currently widely used within the Semantic Web [15]. They aim at structuring the information on the Web in order to make it computer-interpretable. The Web content consists until now for the most part of unstructured text which only humans can read and understand. The syntactical presentation of information on the internet is standardized but not its semantics. However, the search of information on the Web, which is spread over millions of pages, is more successful if it is structured according to its meaning, in other words according to ontologies. Ontology matching algorithms can then automatically find correspondences between semantically related entities of different ontologies for an improved search of the Web. Ontology matching can also find correspondences between heterogeneous product models for an automatic translation of information between these. However, ontology matching only works reliably in special cases and represents an ongoing research effort [40].

2.3.1 Ontology representation languages

Several ontology representation languages with different levels of expressivity are available to formally specify an ontology. Among the many different ontology languages, the Resource Description Framework (RDF), the RDF Schema (RDFS) and the Web Ontology Language (OWL) are very widespread, especially as they belong to the family of specifications of the World Wide Web Consortium[6] (W3C) which is the main international standards organization for the Web. RDF enables to make statements in the form of subject-predicate-object expressions, in other words concept-relation-concept structures, which are known as triples. This corresponds to the AAA slogan stating that "Anyone can say Anything about Any topic" which applies for Web information. As a consequence, the open world assumption stating that information is true until it has been proven false, applies to the Web content as new unpredicted content can be added any time. A collection of RDF statements can be represented as a labeled directed graph, which is called a semantic network. RDFS extends RDF by introducing terms for the classification of RDF data. It is thereby similar to other schema languages. Furthermore, RDFS has been

[6] W3C, `www.w3.org/`

extended to form the OWL by adding new constructs which restrict property values or the number of distinct values for a specific property. OWL is considered the successor of the DAML+OIL[7] ontology language which stands for DARPA[8] Agent Markup Language (DAML) + Ontology Interchange Language (OIL).

The definition of an ontology which includes the choice of ontological categories is required in many disciplines such as designing a database, a knowledge base or an object-oriented system [149]. Ontologies are in this sense similar to conceptual data models. However, the main motivation for defining ontologies is to use them as knowledge bases upon which reasoning procedures can be performed to infer new knowledge. The triples of an ontology are thereby interpreted as first order logic statements, which a reasoner can automatically process to derive new conclusions. The main tasks of reasoners include consistency checking, inference procedures and queries [43]. Common reasoning engines are for example Pellet[9], RacerPro[10] and FaCT++[11].

Most ontologies sacrifice expressiveness in domain modeling in order to achieve in return computational reasoning advantages. The compromise is visible in the different variants of OWL. OWL Full is meant for users who want maximum expressiveness with no computational reasoning guarantees, while OWL DL consists of the full OWL constructs under the condition that the constructs are defined according to some constraints to guarantee computational completeness and the decidability of reasoning systems. Completeness means that all conclusions are guaranteed to be computed and decidability means that all computations will finish in finite time [164]. DL stands for Description Logics [9] which is a family of knowledge representation languages that allow decidable first order logic reasoning. OWL DL supports knowledge representation and reasoning capabilities according to Description Logics. For this sake, OWL DL for example clearly requires a strict separation of classes and individuals, which is not the case with OWL Full. The Knowledge Interchange Format[12] (KIF) and its later version called Common Logic[13] are for example languages which support more features for first-order logic reasoning than Description Logics but at the expense of the decidability and computational efficiency.

In addition to being updated by new facts computed by a reasoner, ontologies can be transformed through the execution of rules which can add new properties or change property values. The Semantic Web Rule Language (SWRL) is a proposal of the W3C to for-

[7] DARPA Agent Markup Language, `http://www.daml.org/`
[8] Defense Advanced Research Projects Agency, `http://www.darpa.mil/`
[9] Pellet: The Open Source OWL Reasoner, `http://clarkparsia.com/pellet`
[10] RacerPro, `http://www.racer-systems.com/products/racerpro/`
[11] FaCT++, `http://owl.man.ac.uk/factplusplus/`
[12] Knowledge Interchange Format (KIF), `http://www.ksl.stanford.edu/knowledge-sharing/kif/`
[13] Common Logic Standard, `http://common-logic.org/`

mally define rules which several reasoners already support. SWRL is also used within the Semantic Query-Enhanced Web Rule Language (SQWRL) to perform queries on OWL ontologies. The other well-known query language, SPARQL Protocol and RDF Query Language (SPARQL), was on the other hand specifically developed for RDF ontologies.

Ontologies are separated into domain and upper ontologies. Domain ontologies, or domain-specific ontologies, model a specific domain while upper ontologies, also called foundation ontologies, are meant to be used across a wide range of domains. Upper ontologies consist of many thousand concepts to represent a structured subset of natural languages like English. In contrast, OWL, which is mainly used to define domain-specific ontologies, consists of around 50 language constructs. Examples of upper ontologies are the Cyc[14] ontology and the Suggested Upper Merged Ontology[15] (SUMO), which is an IEEE candidate for a standard upper ontology (SUO). Although a common universal ontology would be ideal for knowledge sharing, it is not sure whether it will ever exist [109]. The lack of a consensus for a wide coverage ontology is very probably due to its large size and to the different focus and cultural influences of its various contributors. Philosophers ranging for example from Heraclitus to Pierce and Whitehead have developed different categorizations of concepts [149] and it is highly probable that debates on this topic will continue. It is also questionable how concepts can be classified in a standard upper ontology as their meaning is not graved in stone but instead under constant philosophical scrutiny. Even the meaning of meaning, or in other words the answer to the question "What is meaning?", is explained according to different theories by linguists, philosophers [133] and computer scientists [59]. Confronted with apparent vagueness in the definition of concepts, the legitimacy of a standard upper ontology seems fragile [150] and is therefore not further investigated in this work.

2.3.2 Integration approaches

As the role of knowledge in product development increases [157], ontologies have been used in engineering design to formally describe product-related knowledge. Furthermore, ontologies are composed of simple subject-predicate-object expressions which can easily describe relations between different heterogeneous product data sources. The domain-independent ontology modeling languages are therefore employed for the integration of product data from different systems or disciplines [27] .

The construction industry is turning to ontologies for incremental process-driven data integration [139] as the international building information standards do not meet the ex-

[14]Cycorp, Inc., `http://www.cyc.com/`
[15]The Suggested Upper Merged Ontology (SUMO), `http://www.ontologyportal.org/`

pectations of the building sector regarding scope and reaction time for updates, as described in Section 2.2.1. Nevertheless, ontologies do not necessarily replace international standards but may complement them. For this purpose, they are established based upon existing international standards instead of being defined from scratch. The important IFC standard for the building sector has for example been transformed into an OWL ontology in view of supporting ontology-based data integration [14]. Similarly, the prominent STEP, UML and ISO 15926 standards, respectively for mechanical engineering, software engineering and process plants, have been transformed partly into ontologies [93, 48, 12].

Ontologies have been applied in several engineering domains, including CAD systems interoperability [127, 93], assembly design [85], manufacturing processes [27, 99], building design [170], data resources integration [25], mechatronics [31], logistics [50] and product optimization [168]. Most ontologies are described in OWL DL and use SWRL for procedural if-then rules.

Product configuration requires both data integration and knowledge management. Ontologies have therefore often been deployed for product configuration systems [169, 105, 147]. As humans do not think exclusively in hierarchical terms but in associations, the description of product knowledge through semantic networks is often more appropriate than the typical hierarchical structures in PDM systems [29].

Most ontologies supporting product data integration are domain-specific ontologies. However, the Gellish ontological language [138] is a combination of an upper and a domain ontology. It is both very general and domain-specific as it includes concepts related to engineering design. It consists of more than 40000 concepts. Gellish includes concepts such as products, facilities and processes. The name “Gellish” is derived from “Generic Engineering Language”. A subset of Gellish contributed to the development of the ISO 15926 standard for the “Integration of lifecycle data for process plants including oil and gas production facilities”, which is described in Section 2.2.2.

2.4 Summary

Chapter 2 has presented an overview of models which support product data integration. Models for interdisciplinary integration differ in their scope, in their acceptance by peers as an international standard and in their use of generic constructs. Three different approaches are currently pursued to enable multidisciplinary product data integration. The first approach consists of creating new central product models. The second approach supports the development and extension of international standards. The third approach builds upon the expertise of existing standards and combines them with generic concepts

from ontologies or object-oriented modeling to allow a flexible integration of new product aspects.

The development of a new central product model for integration purposes within a specific scenario is a common approach. However, the central product model will eventually need to be adapted based on the evolution of the product itself and on new product data integration requirements. Each modification of the central product model is associated with a lot of effort as interfaces between applications and the central product model then need to be updated.

It is therefore worthwhile to use or develop standards for product data integration which are based on a wide consensus. The large quantity of expertise which flows into the development of an international standard represents to some extent a guarantee for long term stability. The STEP standards have reached a large audience and have greatly contributed to the integration of product information. Standards which cover a single engineering discipline have a predictable scope and have gained wide industry acceptance and stability such as STEP AP203 for the representation of geometric information. In contrast, standards which cover several engineering disciplines, such as in the building and the process plant industries, encompass so many aspects that a standard which could cover them all seems unachievable [37, 139].

In order to describe future unexpected product aspects, product data integration standards offer a backdoor solution by including generic modeling concepts such as *Class*, *Property* and *Object*. These entities are domain-independent and can be used to represent entities of any domain. The domain-specific STEP AP214 and ISO15926 standards have for example included generic object-oriented modeling concepts [101].

Another generic approach consists of using ontologies as a complement to major standards. Ontologies can easily describe relations between different heterogeneous product data resources through simple subject-predicate-object expressions. As a consequence, ontologies are domain-independent and can connect product data from different heterogeneous sources. Instead of redefining an ontology for product data integration from scratch, the terms of an ontology can be imported from well-established international standards. This way, an ontology can reuse a standard and extend it [17].

Chapter 3

UML-based central product model

Standardization efforts have undergone a natural evolution by first focusing on specific domains and eventually on extensions for the integration of more disciplines. However, standards for cross-domain interoperability have not yet gained wide acceptance. There is for example currently no widely accepted standard for mechatronics. This is unfortunate as such a standard would improve the communication and the exchange of product information between different engineering disciplines as well as reduce the costs of interoperability [46]. This chapter first presents in Section 3.1 the need of generic modeling concepts in order to represent in a common central product model the variety of domain-specific modeling concepts. Section 3.2 showcases the common modular structure of product models from different disciplines and thus the necessity for the central product model to represent modular components. Section 3.3 presents the use of the UML to establish a standard central product model as it provides standardized generic modeling concepts which can represent modular components from different engineering disciplines. Section 3.4 compares the generic modeling concepts of the UML with other standardized modeling languages such as OWL and EXPRESS and Section 3.5 presents UML-based product data integration approaches.

3.1 Generic modeling

Central product models need to represent specific model information from various disciplines in order to describe inter-model dependencies. Ideally, the representation in the central product model should be based on commonly used modeling concepts which are familiar to engineers. Discipline-specific engineers can then easily recognize their specific model information within the central product model. However, central product models cannot reuse the modeling concepts of specific models as they are too diverse and

numerous. Even if it were possible to gather all past and present specific modeling concepts, new models with new specific modeling concepts continually appear. Confronted with the constant emergence of specific modeling concepts, the central product model would need to be permanently updated. As a consequence, it would lose its legitimacy as a standard.

Instead of supporting an unlimited number of specific modeling concepts, central product models can consist of a manageable set of overarching generic modeling concepts which correspond to various specific modeling concepts. Generic modeling concepts are for example a *Module*, an *Instance* or a *Property*. Each domain can be represented on an abstract level through a reduced set of generic modeling concepts, as depicted in Fig. 1.4. This is for example common practice in data modeling as shown in Fig. 2.1. As a consequence, the non-solvable problem of including an unlimited number of specific modeling concepts in the central product model can be transformed into the solvable problem of supporting a limited number of corresponding generic modeling concepts. As described in Section 2.2, the developers of integration standards in the multidisciplinary building and process plant industries had difficulty in specifying in advance all required specific modeling concepts. As a consequence, both sectors turned to generic modeling concepts to easily capture information, which was unforeseeable during the development of the standards [139, 95].

In order to be used for interoperability across most disciplines, a central product model must include generic modeling concepts which enable to clearly represent a maximum number of specific modeling concepts. For example, a mass flow or a data flow cannot be clearly described through an *Object* entity as it represents a static artifact. In this case, a generic *Information flow* entity would for example be better suited. If the set of generic modeling concepts of the product model is too small, the central product model cannot easily and precisely represent deep nested specific model information. If on the other hand, the set of generic modeling concepts is too large, the central product model may be too difficult to understand and confusion is probable. No set of generic modeling concepts for a standard central product model has yet been widely adopted.

3.2 Modeling modular components

In spite of the diversity of engineering models, models which are defined with state-of-the-art modeling applications share common modeling concepts in order to support a common modeling requirement which is modularity. Independent of the engineering discipline, models can reach in complex projects large sizes which are hard to overview.

In order to reduce the complexity and enable the reusability of model components, models are decomposable into exchangeable, maintainable and reusable model components, in other words modules. The reusability of model components facilitates model updates and the design of a variety of model configurations. In contrast, a lack of modularity leads to time-consuming and expensive single purpose models which need to be developed from scratch. The benefits of modularity not only apply for modeling but naturally also for the manufacturing of real products [160]. According to Baldwin and Clark, the manufacturing industry has reached high levels of innovation and growth through the design of complex products from smaller independent modular components [10].

Modules commonly have in- and outputs. As a result of the decomposition of a model into distinct encapsulated model components, model components within a model are often interdependent. Modules therefore specify their required inputs and provided outputs, based on which the compatibility of components, when replacing a model component with another or placing a model component within a specific model context, can be automatically checked. The automatic compatibility check of components is especially useful in large models with many components and dependencies.

In addition to specifying in- and outputs, modules follow the principle of information hiding by means of a clear distinction between their internal information, which is hidden from other components, and their outward-facing information which is visible to other components. The compatibility of model components only depends on their outward-facing information, which includes their in-and outputs. The distinction between visible and hidden module information is necessary in order to distinguish between the hidden information, which may change over time, and the visible information, which needs to stay constant in order to support compatibility with other modules. If a CD were a module, the content of the CD would represent the hidden module information, as it may change over time, while the geometric form of the CD would represent the visible module information, which may not vary over time in order to support compatibility with CD drives.

Other common modeling concepts to describe modules are the notions of a module template and a module instance. The description of similar modules is simplified by defining a module template which specifies the module features which are common to a group of modules, or in other words a specific module type. The description of a module based on a template takes advantage of the once predefined template features instead of redefining them from the ground up. The description of a module based on a module template is most often called a module instance.

The multidisciplinary design of mechatronic products often involves software, geometric and dynamic systems modeling. In software design, the notion of a software mod-

ule has evolved from a function in procedural programming to an object in object-oriented programming. A software object thereby encapsulates functions and related variables. The main software module template is called a class while the module instance is termed an object. The provided and required information of a software component is specified through interfaces. The concept of an interface is also used to specify the outward-facing visible information of a software object, in other words the features of a software object which are accessible to other software objects. The terminology to describe object-oriented software is uniform and mostly independent of a specific programming language.

In geometric modeling, the most common modular unit within CAD models is a part which represents a grouping of geometric information. A part also represents a geometric template which can be instantiated and inserted into various geometric assembly models. The terminology to describe the provided and required information of parts is not uniform but specific to the CAD software. In CATIA[1] for example, dependencies between parts are described as import or cut-copy-and-paste links and the information which a part makes available to other parts is referred to as published features.

Modular units in dynamic system models are often called blocks whereby block types correspond to module templates and block instances to module instances. The terms used to specify blocks and their provided/required information are specific to the dynamic system modeling software. In a dynamic system modeling application such as Simulink[2], the provided/required information of blocks is specified through block ports. Furthermore, the blocks can be used in models as black boxes whereby their internal information is hidden and only their ports are visible.

A multitude of different generic modeling concepts have been developed to describe modular architectures for product family design and platform-based product development [78]. Graph-based representations of products, in which product modules are described as *Nodes* and interactions as *Edges*, have been used in combination with graph grammars to automatically design specific product variants based on production rules. Graph grammars were applied among others to the design of coffee-makers [146], power supplies [36] and satellites [144]. Männistö and Sulonen [102] proposed an approach based on generic data modeling concepts to describe configurable products. The approach used modeling concepts such as *Class*, *Attribute* and *Inheritance* to represent the characteristics of product families and also *Individuals* to describe concrete products within a product family. Paredis et al. [123] defined modeling concepts such as *Component objects*, *Ports* and *Connections*, to describe the decomposition and interactions of mecha-

[1] DASSAULT SYSTEMES, `http://www.3ds.com/products/catia/`

[2] The MathWorks, Simulink,
`http://www.mathworks.com/products/simulink/`

tronic systems. *Component objects* represented both CAD and behavior models, whereby *Ports* specified the interaction between different domain-specific subsystems.

Although concepts to support modularity are frequent across several modeling disciplines in engineering design, no standardized vocabulary has been adopted to describe their common modular structure. Due to the variety of discipline-specific terminologies and the widespread marketing policy of software providers boasting with so-called exclusive modeling features, the modeling concepts to describe modular model components do not share the same terminology.

3.3 UML-based object-oriented modeling

In software design, object-oriented programming and modeling concepts have contributed to higher modularity. Programming concepts have been developed in software engineering to support modularity, whereby the most prominent are object-oriented programming concepts which consist in encapsulating variables and functions into modular units called objects. Object-oriented software models describe the classification, communication and internal structure of objects within a software architecture.

Although object-oriented modeling is currently mainly used for software modeling, object-oriented modeling concepts are generic and can be used to describe various modular structures. As the term object already suggests, an object can represent a software object as well as a physical or model component. Object-oriented modeling is thus not restricted to software modeling. The Modelica modeling language for example uses the concepts of object-orientation to describe dynamic system models which include for example mechanical, electrical, hydraulic, thermal and control components [45]. The generic object-oriented modeling concepts of software modeling can thus be reused in the context of a central product model to represent the common modular structure of various discipline- and application-specific models.

As a central product model is meant to be used across several disciplines, it addresses many parties and therefore requires standardization. Any proposed generic modeling concept will only have a universally accepted meaning through standardization effort. Ideally, the generic modeling concepts of a central product model should therefore already be standardized.

The Unified Modeling Language (UML) is the de facto standard for object-oriented modeling and is a general purpose visual modeling language used to specify, visualize, modify, construct and document software. UML concepts are based on the object-oriented programming paradigm. Section 3.3.1 describes the evolution of programming paradigms

which led to object-oriented programming and ultimately to object-oriented modeling techniques. Section 3.3.2 presents the UML and its role within software engineering while Section 3.3.3 introduces the UML specification. Section 3.3.4 showcases the main UML modeling concepts which are used in this thesis for the representation of geometric, dynamic and multibody system model information.

3.3.1 Origins of object-oriented software development

Programming paradigms have evolved over time towards human-friendly higher levels of abstraction and away from the machine-friendly low-level bits. In the 1940s, the focus was on reducing hardware costs and the art of software programming was neglected. Programming was at that time very low-level and consisted of machine code equivalent to a sequence of bits. Assembly languages were developed in the 50s to replace machine code by more human-readable symbolic labels composed of letters and decimal numbers.

The next major step towards higher productivity in software programming was realized through procedural languages, also called high-level programming languages, which introduced more abstract language constructs such as arithmetic expressions, statements, arrays and subroutines. FORmula TRANslation (FORTRAN) was in 1957 among the first procedural languages. Many low-level memory operations were automatically managed by compilers. An operation such as i+j was executable without specifying where to store i and j in memory and what machine instructions were needed to retrieve and add them. The productivity of programmers increased significantly as they could concentrate on their design intent and ignore low-level repetitive memory management tasks.

Although it became easier to write code, programming was a creative art and the result was often hardly understandable code which was qualified as spaghetti code. Consequently, software was too complex to be maintained and software development too costly. In 1968 the term “software crisis” was used to highlight these problems. Solutions for a systematic software development process were presented under the umbrella of a new discipline named “software engineering” [107].

Improvements emerged through concepts related to structured programming. A major source for unreadable spaghetti code was the goto statement which enabled to jump from one line of code to any other. After it was proven that equivalent code could be written without that statement, its use was considered in 1972 harmful and not recommended [33]. This had an effect on the graphical representation of programs. Traditional flowcharts were replaced in 1973 by structured flowcharts [106], also called Nassi-Shneiderman diagrams or structograms, which did not include lines to represent goto statements as they were to be avoided. In addition, new analysis methods appeared to capture program re-

quirements in order to avoid future costly program updates. Data flow diagrams, data dictionaries, structured English, decision tables and decision trees were used instead of conventional narrative text to clearly define the program specification and facilitate the transition from program analysis to program design [32].

The flexibility and comprehensibility of programs was largely dependent on the choice and design of reusable software modules. The program decomposition into reusable software modules was first based on flowcharts. This conventional decomposition technique ensured a separation of concerns between different modules to avoid any overlapping in functionality. In 1972, a new decomposition technique based on information hiding was introduced by Parnas [124]. It was better adapted for future software updates than the conventional decomposition based on flowcharts. The new decomposition strategy created software modules based on the functions which were likely to change and not on the high-level program functions which were derived from the flowchart. Software modules were conceived to be easily exchangeable by sharing the same interface. Changes in the interfaces were to be avoided as they triggered time-consuming changes in the software modules and in the programs which referred to the software modules. The concepts of information hiding helped in designing long-lasting interfaces by hiding the changeable or highly detailed information in the software modules while their interfaces only contained long-lasting information. In procedural programming, software modules were procedures and software module interfaces were equivalent to the input and output arguments of procedures.

Although the procedures were designed to be easily exchangeable and reusable, they did not declare their side effects on global variables. The supposedly independent procedures were indirectly linked if they shared the same global variables. Furthermore, the reuse of a procedure within another context without the required global variables was a source of error. The undeclared dependencies between procedures and global variables often led to unexpected effects and hampered the reuse and exchange of procedures.

This was avoided by encapsulating procedures and their required variables into objects. This data encapsulation represented the next software modularization step [167]. Procedures and their affected variables were thereby regrouped into single units called objects. Figure 3.1 presents the transformation from procedure-based to object-based software modularization through data encapsulation. Procedures automatically had access to their required variables and objects clearly declared the dependence between the procedures and their required variables. As a result, the reuse of procedures in combination with their required variables had no unexpected side effects. Global variables were then limited to object-independent variables such as constant values.

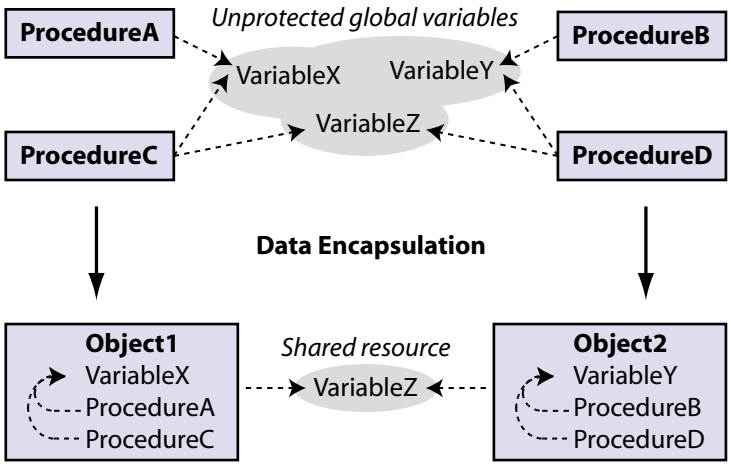

Figure 3.1: Transformation from procedure-based to object-based modularization through data encapsulation (after Wegner [167])

However, objects could still access freely and in an uncoordinated way each other's variables, which led to unexpected and wrong variable values. The concept of object interfaces was introduced to control the encapsulation of data within objects. The interface of an object described the information of an object which was visible and accessible to other objects. The other information, which was inaccessible or invisible to other objects, formed the object's black box. The definition of invisible variables guaranteed that objects could not interfere freely in each other's variables. The access of objects on each other's variables, which in an uncontrolled way was a source of error, was controlled by object interfaces.

The general concepts of information hiding were also applied to the design of object interfaces for an improved software modularization. An interface can theoretically be composed of both variables and procedure signatures. Under the condition that the interface does not change over time, objects can be easily exchanged if they share the same interface. The principle of information hiding stated that software module interfaces are long-lasting if they do not contain changeable or highly detailed information. In regard to objects, this meant that object interfaces could not contain variables as they are detailed and most likely to change. The implementation of a procedure could for example change over time and require another set of variables while the procedure signature stayed identical. Figure 3.1 shows the transformation of object interfaces into long-lasting interfaces containing only procedures. As a result, objects could only retrieve or change each other's variables indirectly by passing through intended procedures. The execution of object-oriented programs does not consist of a sequence of procedures as in procedural programs, but of an interplay of modular software objects.

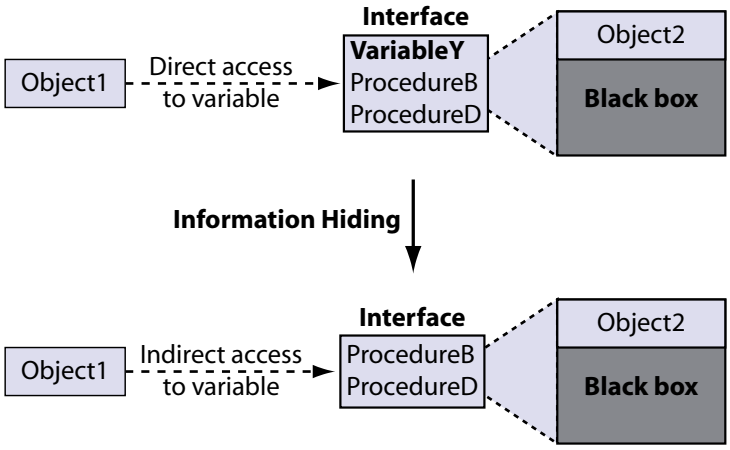

Figure 3.2: Transformation of object interfaces into interfaces containing only procedures in view of improving software modularization according to the principles of information hiding

Many objects sharing the same variables and procedures neeeded to be created so classes were introduced as templates for objects. Procedures inside classes are usually called operations or methods. Figure 3.3 for example shows a Circle class as template and derived circle objects acting as template instances. Classes describe the variables and procedures which are common to a group of objects while objects store attribute values.

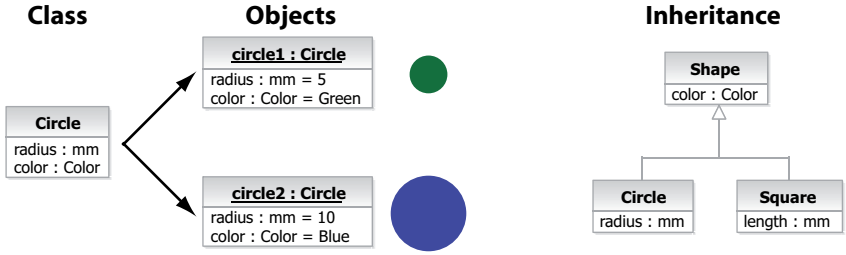

Figure 3.3: Left: Classes as templates for the creation of objects. Right: Inheritance hierarchy between classes

Further, the concept of inheritance was introduced to create new classes by reusing existing class definitions. Code duplication was thereby prevented. Figure 3.3 for example shows the Circle and Square classes which inherit the properties of the Shape class. This avoids the redundant declaration of the color attribute in the Circle and Square classes. Programming languages which supported modularization in objects and classes were called object-based and those which in addition included inheritance object-oriented [140]. Simula was in 1967 the first object-oriented programming language. These object-oriented principles were taken over by future programming languages such as Smalltalk, Ada, C++ or Java.

3.3.2 UML for software engineering

Similar to the rise of structured programming concepts after the emergence of procedural programming languages, many methodologies emerged following the rise in popularity of object-oriented programming for a systematic design of object-oriented software [60]. Among the most prevalent were the Object Modeling Technique (OMT) of Rumbaugh et al. [143] in 1991, the Object-Oriented Design (OOD) of Booch [20] in 1994 and the Object-Oriented Software Engineering (OOSE) method of Jacobson et al. [77] in 1992. Each methodology had its own set of terminology, notation and specific focus but there was a pool of common core concepts. The first attempts to unify these methods, such as Fusion by Coleman et al. [28], did not succeed as they did not involve the developers of the original methodologies. Ultimately, the three methodologists James Raumbaugh, Grady Booch and Ivar Jacobson, later referred to as the three amigos, overcame their differences and conceived in a joint effort in 1997 the Unified Modeling Language (UML) [21]. Although the UML does not prescribe a specific software design process, it can be combined with different software development frameworks such as the Unified Software Development Process [76] or the Model Driven Architecture [112].

The UML offers a unified representation of an object-oriented software architecture and plays an essential role in the communication between software engineers, similar to the role of a blueprint between mechanical engineers or architects. The UML, as a formal modeling language, enables to better bridge the gaps between software requirements, analysis and coding than arbitrary diagrams and text. The classification, composition and communication of objects are described graphically through modeling languages, of which the UML is the most well-known. The UML consists of 13 diagram types for the description of structural and behavioral object aspects. Table 3.1 presents the list of diagrams, their main modeling concepts and their classification in structural or behavioral diagrams. Figure 3.3 for example presents the class, object and inheritance concepts in class diagram notation. A detailed description of the UML diagrams used in this thesis are presented in Chapter 3. A UML model is usually defined by several diagrams, whereby elements of different diagram types can be interlinked. The evolution and standardization process of the UML is managed by the Object Management Group (OMG) which regularly issues new UML versions. The UML version 1.4.2 was also released as an ISO international standard [74] in 2005.

The UML is often used as a basis for code generation in model-driven software engineering frameworks, such as the Executable UML approach of Mellor et al. [103] in 2002 or the Model Driven Architecture (MDA) [112] initiative from the Object Management

Major Area	Diagram Type	Main Concepts
Structure	Class diagram	class, association, generalization, interface
	Composite structure diagram	part, port, connector, role, collaboration
	Component diagram	component, port, dependency, realization
	Deployment diagram	artifact, node, deployment
	Object diagram	instance specification, link
	Package diagram	package, package import, package extension
Behavior	Activity diagram	activity, action, node, flow
	Sequence diagram	interaction, lifeline, message, occurrence
	State machine diagram	state machine, state, transition, region
	Use case diagram	use case, actor, extend, include
	Communication diagram	interaction, lifeline, message, sequence number
	Interaction overview diagram	interaction, interaction use, activity node
	Timing diagram	interaction, lifeline, state, timeline, duration

Table 3.1: UML 2.1.1 diagrams

Group[3] (OMG). Modeling languages such as the UML are used in this sense not only for documentation but for code generation and ultimately for programming. Modeling languages thereby potentially represent a new programming paradigm [56] at a higher abstraction level than textual programs as shown in Fig. 3.4. However, the transformation of a graphical model such as a UML model into a lower level textual program is not yet as mature as the similar compilation of a high-level program such as C into lower level assembler code or machine language.

A UML model can represent a software architecture at different abstraction levels. A software representation in UML can for example be independent of the programming language in which the software is coded and independent of the operating system on which the software runs. On the other hand, a UML model can also include programming language- and operating system-dependent features. Within the MDA approach of the OMG, the abstract UML model is called Platform Independent Model (PIM) while the less abstract UML model is called Platform Specific Model (PSM). A PSM is derived from a PIM within the MDA software development lifecycle as displayed in Fig. 3.5. As the same software often needs to run on different operating systems, the separation in PIM and PSM enables to reuse the PIM for several specific platforms. Several PSMs can be generated based on one PIM through model transformations. The OMG issued in

[3]Object Management Group (OMG) in 2003, `http://www.omg.org/`

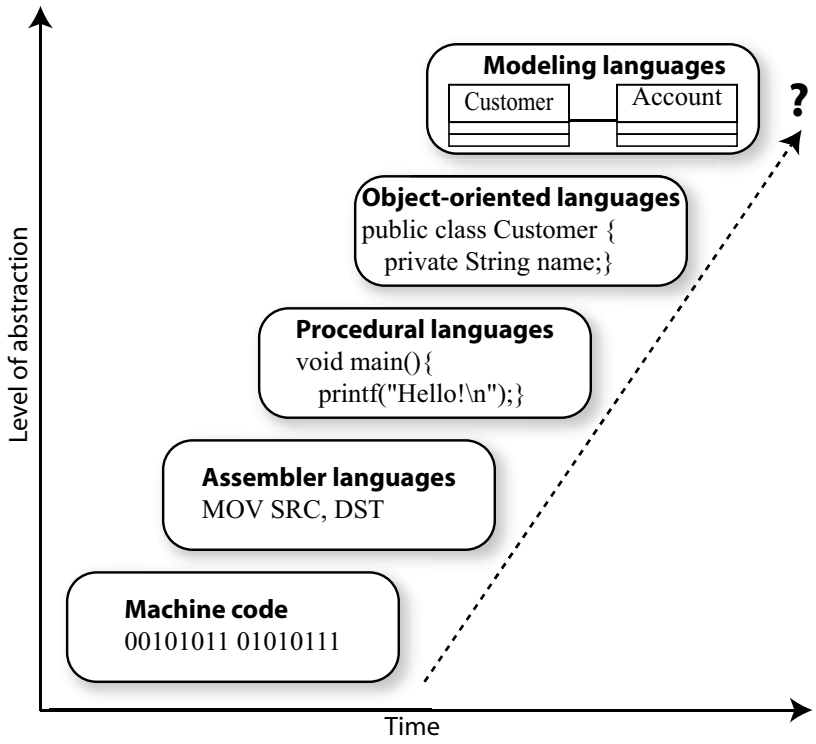

Figure 3.4: Increasing levels of abstraction in programming paradigms (after Gruhn [56])

2008 the Query/View/Transformation (QVT) standard [119] to uniformly describe model transformations.

3.3.3 UML specification

The syntax and semantics of the UML are specified in the UML Infrastructure and Superstructure specifications [121, 122]. The UML syntax is divided into concrete and abstract syntax similar to the syntax specification of programming languages [104]. The concrete keyword-dependent syntax of programming languages is specified by grammars in Backus-Naur Form (BNF) while the concrete graphical UML syntax is specified through style guidelines described in English and through graphical examples. The UML Class construct for example is to be represented through a rectangle which should optionally contain compartments for attributes and operations. On the other hand, the abstract UML syntax describes the valid relationships between UML constructs and how the UML constructs are built up. Similarly, the abstract syntax of programming languages describes the relationships between programming language constructs such as Program, Declaration and Variable in a keyword-independent way.

The abstract UML syntax is described by the UML metamodel. It is composed of a subset of the UML which is called the Infrastructure Library [121]. It contains con-

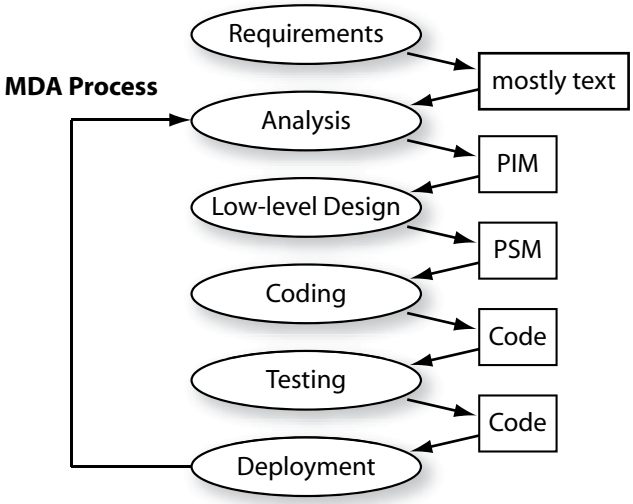

Figure 3.5: MDA software development lifecycle (after Kleppe et al. [87])

structs of UML class diagrams which are very general such as Class, Association and Property. The Infrastructure Library is therefore reused at a higher abstraction level to define the UML meta-metamodel which is specified by the Meta Object Facility (MOF) Core Specification [113].

In theory, an unlimited number of modeling layers, or metalevels, can exist. No additional layers above MOF are defined because it is reflexive due to its self-defining elements. The same bootstrapping technique also applies to the Extended BNF (EBNF) notation whose self-representation in EBNF only takes some lines [52]. MOF is the metamodel for many metamodels other than the UML metamodel. MOF version 1.4.1 was released in addition to the OMG as an ISO standard [75] in 2005. MOF and MOF-based models are serialized in XML according to the XML Metadata Interchange (XMI) standard [117]. As a consequence, different modeling tools can exchange UML models as they are serialized in the common XMI interchange standard.

The semantics of the UML are described through formal constraints expressed in the Object Constraint Language (OCL) [114] and in precise English. The detailed semantics of each UML construct are described in English. A list of constraints applies to each UML construct to define additional well-formedness rules. Constraints are formally defined in OCL when possible. Overall, the UML specification is composed of a combination of languages including a subset of the UML, the Object Constraint Language (OCL) [114] and precise English. The definition of the UML is summarized in Table 3.2.

Modeling tools are not forced to support the complete UML specification. Compliance levels and language units have been defined in order to enable modeling tools to

Concept	Purpose	UML solution
abstract syntax	the concepts from which models are created	class diagram at level M2
concrete syntax	concrete rendering of these concepts	UML notation informally specified
well-formedness	rules for the application of these concepts	constraints on the abstract syntax (e.g. using OCL)
semantics	description of the meaning of a model	natural language specification

Table 3.2: UML Language Definition (after Atkinson and Kühne [8])

support and share the same UML subsets. Four compliance levels containing different language units have been defined, whereby each upper level comprises and extends the functionality of the lower level. The lowest compliance level is L0 and only includes language units related to classes, types and packages. In order to exchange UML diagrams between tools, the Diagram Interchange specification [115] has been defined. The adherence to a compliance level is further detailed by expressing the adherence to the abstract syntax, concrete syntax and the Diagram Interchange standard. The abstract syntax compliance for example enables to output and read UML models in XMI. Concrete syntax compliance enables to represent UML constructs in the standardized notation.

3.3.4 UML modeling concepts

This Section presents the most important UML diagrams which will be presented in Sections 4 to 6 to describe application-specific model information from different disciplines in a UML-based product model. Each UML diagram only represents a specific aspect of a UML model. As a consequence, a UML model usually consists of several UML diagrams which complement each other. In this Section, UML class, composite structure and activity diagrams are introduced. The UML consists of over 200 modeling concepts of which only a few dozen are used frequently. The complete UML reference is the UML specification which consists of the UML Infrastructure [121] and UML Superstructure [122] documents.

The essential UML modeling concept is a *class*. A class declares the features, constraints and semantics which are common to a group of objects. A class acts as a template from which concrete objects can be instantiated. Objects are synonymous with template instances or with the UML modeling construct *instance specification* and can theoretically stand for anything, whether software or physical entities. Depending on the domain-specific context, templates and template instances appear under different terms.

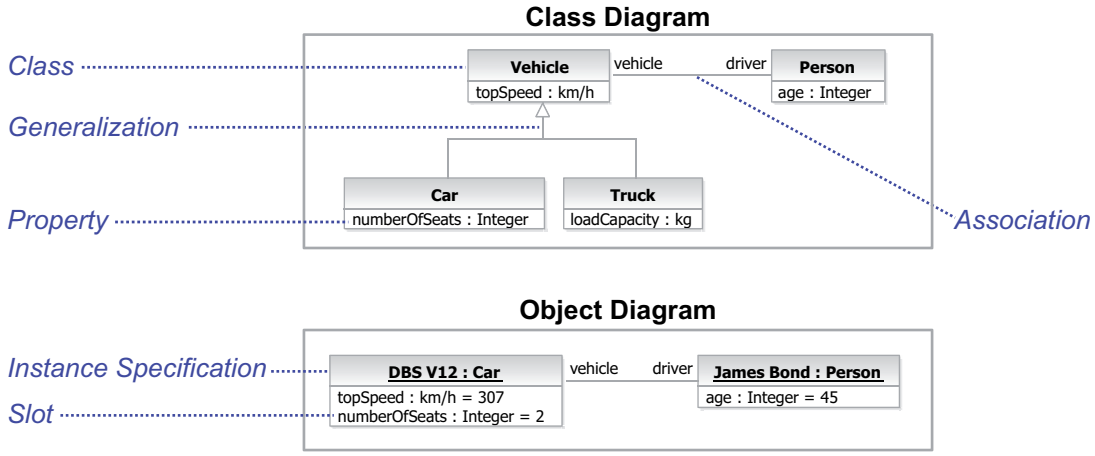

Figure 3.6: UML class and object diagrams

The class and object diagrams in Fig. 3.6 for example include a “Car” class and a corresponding “DBS V12” instance. The classifier of the instance is shown next to the instance name. Attributes of a class which are common to all instances of a class are described in UML as *properties*. The “Car” class for example has the “numberOfSeats” property of *type* “Integer”. The instantiation of properties is described through *slots* . The “DBS V12” car instance for example has a slot referring to the “numberOfSeats” property with a value equal to two.

An important concept in object-oriented modeling is the inheritance relationship between classes. A child class thereby automatically inherits the properties of a parent class. Redefinition of identical properties in a similar class is thereby avoided. The concept of inheritance enables to categorize classes in hierarchies based on their similarities. The relationship between the child and the parent class is termed *generalization* in UML. The “Car” class in Fig. 3.6 for example has a generalization relationship with the “Vehicle” class. The “topSpeed” property of the parent “Vehicle” class is thereby transmitted to the child “Car” class. As a consequence, the “DBS V12” car instance has a slot referring to the “topSpeed” property with a value equal to 307km/h.

A class property can be represented graphically through a line going from the class owning the property to the class being the property type. The line is termed *association* in UML and allows to visualize class dependencies. Property names and multiplicities are then depicted next to the associations. The “Vehicle” class in Fig. 3.6 for example has a “driver” property of type “Person” which is represented graphically through an association between both classes. Similarly, the “Person” class has a “vehicle” property of type “Vehicle”. This type of association is bidirectional and is equivalent to two separate directed associations. The association can be instantiated in the object diagram to

represent the instance slots which correspond to the class properties of the association. The association instance is thereby described by a line called a *link* which connects the related instances. The "James Bond" person instance for example has a slot referring to the "vehicle" property with a value equal to the "DBS V12" car instance and vice versa the "DBS V12" car instance has a slot referring to the "driver" property with a value equal to the "James Bond" person instance.

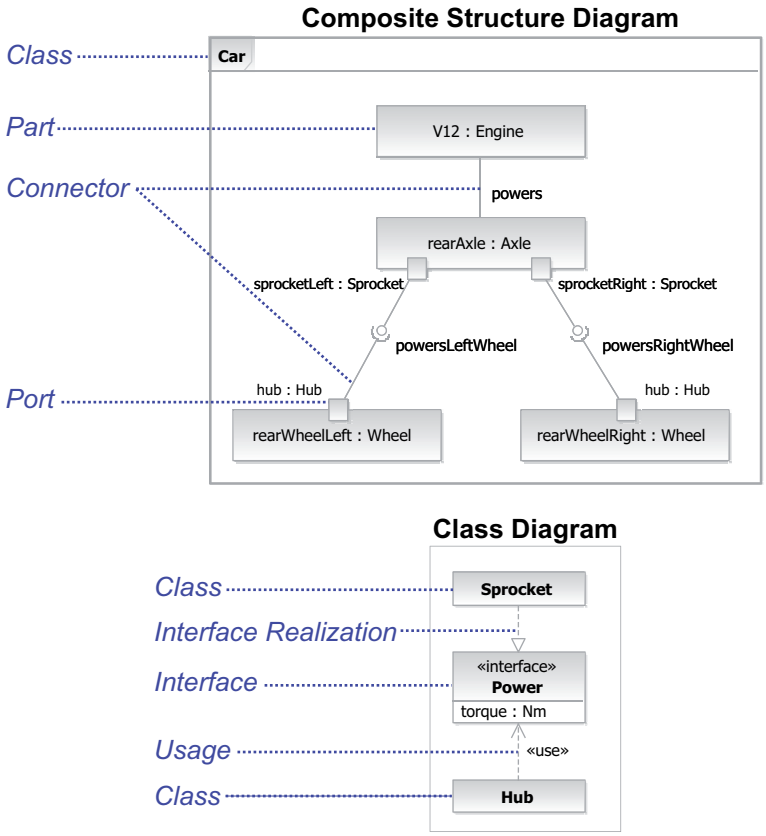

Figure 3.7: UML composite structure and class diagrams

The composition hierarchy within an object is represented through the composite structure diagram of its classifier. The internal structure of a car is for example represented through the composite structure diagram of the "Car" class as in Fig. 3.7. A composition structure diagram shows internal objects and their relationships. The composite structure diagram does not directly specify internal instances within a composite object but the roles that internal instances play. Properties of the composite class which correspond to roles are called *parts*. The "Car" class is for example composed among others of the "V12" part of type "Engine" and of the "rearAxle" part of type "Axle". The destruction of the composite object leads to the destruction of its internal objects.

Relationships between roles are described through *connectors* and only apply within the context of the composite object. Connectors describe links or the exchange of information between parts. A link may be an instance of an association, or it may represent the ability of the instances to communicate. The connector named “powers” between the “V12” and “rearAxle” parts for example represents a link. A more detailed description of the information exchange between parts requires *interfaces* and possibly *ports*.

Interfaces are protocols for the exchange of information. Parts can be connected with each other if their interfaces are compatible. Ports describe the possible interaction points of parts. A port can be linked through a connector with another port if the required interfaces of one port are provided by the other and vice versa. The “rearAxle” and “rearWheelLeft” parts are for example connected through their respective “sprocketLeft” and “hub” ports which share the common “Power” interface, as depicted in the class diagram in Fig. 3.7. The “Sprocket” class implements, or in other words delivers, the information which is specified in the “Power” interface while the “Hub” class requires it. These dependencies are respectively described in UML through *interface realization* and *usage* relationships, as shown in the class diagram in Fig. 3.7.

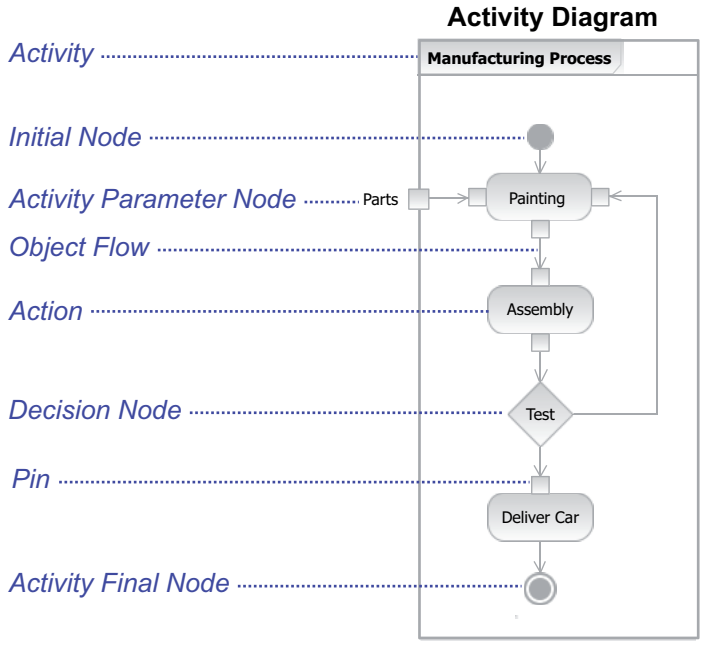

Figure 3.8: UML activity diagram.

Flow graphs are generally used to depict a certain logic or process. Flow graphs are described in UML through *activities* composed of *nodes* and *edges* and are displayed as activity diagrams. Activity nodes represent *actions*, input and output objects of actions in the form of *pins* or the coordination of flows through for example *fork nodes*.

A “Manufacturing Process” is for example described as UML activity diagram in Fig. 3.8. The activity starts at the *initial node* and finishes at the *activity final node* and is for example composed of the “Painting” and “Assembly” actions. Activity edges represent directed connections between activity nodes. *Object flows* are edges which have objects passing along them. Activities can have input and output parameters through *activity parameter nodes*. The “Manufacturing Process” activity for example has a “parts” input parameter which is sent to the “Painting” action through an object flow.

The UML concepts of classes, instances, relationships, composite structures and activities are generic. The UML can therefore also be used to model non-technical aspects. Parunak and Odell [126] have for example represented social structures such as terrorist organizations through UML class and activity diagrams. Next to the use of the UML to describe object-oriented software architectures, UML class diagrams are typically used to describe data models.

The featherweight extension mechanism is the easiest and consists of adding keywords onto the general purpose UML modeling constructs. The lightweight extension mechanism stands for the attachment onto the UML elements of *stereotypes* which for further detailing can own properties. Stereotypes enable to add domain-specific information on top of the generic UML modeling elements. Several stereotypes can be applied to the same UML modeling element. Stereotypes are graphically represented within a pair of guillemets. Stereotypes are for example applied to a class and its properties, as depicted in Fig. 3.9, to represent a CATIA-based geometric part and its attributes. They can also have an icon to change the graphical appearance of the UML element it is applied on. This increases the recognizability of the specific model information within the general purpose UML model.

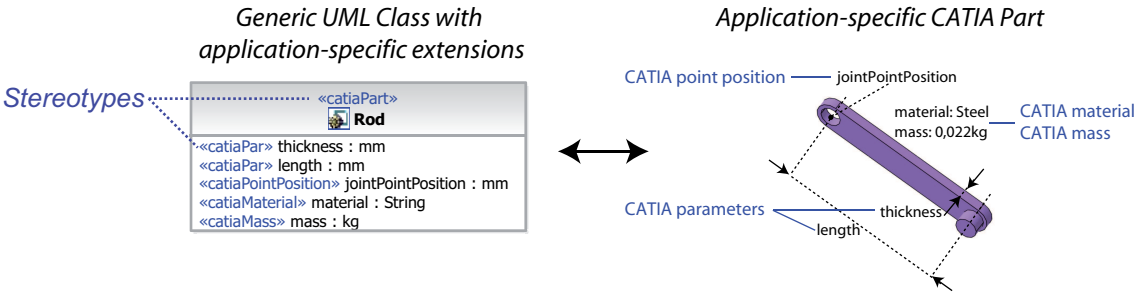

Figure 3.9: Example of UML stereotypes

The stereotypes specific to a domain are regrouped in a special UML *package* called *profile*. All geometry-specific stereotypes of Fig. 3.9 would for example be defined in a geometry-specific profile. Several profiles are for example available for the design of real-time embedded systems [120], airworthiness-compliant safety-critical software [173]

or System on a Chip (SoC) [116, 145]. However, no profiles have been developed to represent geometric, controller and multibody system model information in UML.

Heavyweight extensions aim at modifying or creating UML modeling constructs. It is generally recommended to favor the slimmest possible UML extension in order to share the highest interoperability with other UML users and tools. The lightweight approach was therefore chosen as UML extension mechanism in the context of this thesis.

3.4 UML for product data integration

Important modeling languages such as EXPRESS for the standardization of product data, UML for the specification of software architectures and OWL for ontologies consist of a wide range of generic modeling concepts which have gained wide acceptance. Among the languages with standardized generic modeling concepts the UML is unique as it is:

- the most widely adopted
- the de facto standard to describe object-oriented systems
- the only one to describe both in detail static and behavioral aspects

The UML, as a de facto standard for object-oriented software modeling [88], is already well established and adopted by many software engineers. Software engineering is not only a major discipline but is playing an increasingly prominent role in product design, especially as more and more traditional mechanical control devices are replaced with electronic devices. The electronic fly-by-wire control mechanism for example, which was first introduced for fighter jets, is now increasingly emerging for higher safety in other types of vehicles such as small aircraft, automobiles and trucks [137].

As described in Section 3.3.2, the UML resulted from the unification of various object-oriented modeling methods based upon the object-oriented programming paradigm. Although it was developed for software modeling, it is considered a general purpose modeling language as it mostly consists of generic modeling concepts.

The UML consists of modeling entitites which can describe in detail both static and behavioral aspects. EXPRESS, like most data modeling languages such as entity-relationship diagrams, can on the other hand only describe a static snapshot of an information model [3]. Similarly, OWL has no language constructs to describe dynamic processes. However, the ability to specify dynamic processes is important in many disciplines such as controller or software design. As described in Table 3.1 in Section 3.3.2, the UML consists of several diagram types, such as activity diagrams, sequence diagrams and state machine diagrams in order to describe dynamic processes. In addition, the UML can

describe the local behavior of entities through methods which only operate on local entity attributes.

Further important UML language features include:

- the modeling of entity instances
- its extension mechanisms
- its non-proprietary freely accessible specification

The UML supports the description of entity instances, which is not the case with EXPRESS or most data modeling languages which strictly separate the concepts of an entity and an entity instance. As opposed to conventional data modeling techniques, the schema and the schema instances can be situated in the same UML model. This allows to quickly change the schema and adapt the schema instances accordingly. The UML does not require the schema instances to conform to the schema. This can be practical in scenarios in which both the schema and the schema instances rapidly change, such as in the modeling of product configurations [102]. This allows to first define instances and to classify them at a later stage for example through reasoning programs as with ontologies or through formal concept analysis [5].

The UML would simply serve a documentation purpose if it only described object-oriented models through general purpose UML modeling constructs. Through the addition of domain-specific information via UML extension mechanisms, the UML model can be interpreted and reused for different purposes. The most frequent example in software engineering is the interpretation of the same UML model for the automatic generation of code in different programming languages. Similarly, the content within a UML model can also be interpreted in order to generate or update specific product models.

The UML is defined through a non-proprietary specification which can be accessed at no cost. This promotes its diffusion among users and software providers. Several open-source projects, such as the Eclipse UML2[4] project, the Eclipse Graphical Modeling Framework[5] and TOPCASED[6] offer free UML editors. In addition, the non-proprietary UML specification enables a company to have the guarantee that it will be able to represent its product information across a complete product lifecycle, which may last for example in the aerospace industry up to 50 years.

[4]Eclipse UML2, `www.eclipse.org/uml2`
[5]Eclipse Graphical Modeling Framework (GMF), `www.eclipse.org/gmf`
[6]TOPCASED, `www.topcased.org/`

According to Szykman et al. [156], a common product information representation for cross-domain interoperability needs to fulfill the following requirements:

- not tied to a single vendor software solution
- open and non-proprietary
- simple and generic
- extensible by allowing additional concepts to create a broader engineering context
- not dependent on any one product development process
- capable of capturing that portion of the engineering context that is most commonly shared in product development activities

The first five criteria are fulfilled by the UML. The UML is an open non-proprietary modeling language which is not tied to any software vendor. It has gained wide acceptance due to its simplicity and generic capabilities and is therefore used for data modeling or ontology modeling next to software modeling. The UML offers extension mechanisms and was designed as an object-oriented modeling language independent of any software or product development process. The presented UML characteristics seem to ensure a long term viability for the UML standard.

According to Szykman's last criteria, a common product information representation should enable interoperability by representing the most commonly shared information. As the UML includes standardized and well-known generic modeling concepts to describe object-oriented systems, this thesis investigates the reuse of the UML to establish a central product model in order to integrate typical mechatronic application-specific model information.

3.5 UML-based integration approaches

The UML offers extension mechanisms in order to reuse the general purpose modeling language for the description of domain-specific information. The same UML class can for example be used to describe a Java or a C++ class. The extension mechanisms are separated into lightweight and first-class extensions. The lightweight extensions, also called stereotypes in the UML terminology, add supplementary semantics to UML elements. The lightweight extensions which are specific to a certain domain are regrouped in packages called profiles. Several profiles are for example available for the design of real-time embedded systems [120], airworthiness-compliant safety-critical software [173] or System on a Chip (SoC) [116, 145]. A new modeling language based on lightweight extensions of the UML is the Systems Modeling Language (SysML) [118] for systems

engineering. On the other hand, the first-class extension mechanism consists of changing the UML metamodel according to MOF. As a consequence, a new modeling language can be created by changing the syntax and semantics of the UML.

The UML is both involved directly in product data integration approaches as an object-oriented modeling language and indirectly as a data modeling language. The use of the UML as a data modeling language, similar to the STEP-related EXPRESS modeling language, is very frequent. Among many examples it was used to define an interchange format for the exchange of STEP- and PDM system-related data [111] or to define a format for electro-mechanical assemblies [134]. Another indirect approach is the use of the UML as a software modeling language to specify systems supporting product data integration. The UML is for example applied to the specification of PDM systems [41, 58].

The direct use of the UML for product data integration is less common as it is still mostly used for the design of software and real-time systems [34]. However, the same adaptation mechanisms which are applied to add domain-specific semantics to the general purpose UML in the context of model-based software engineering can also be used to integrate product information.

The Methodology for Knowledge Based Engineering Applications (MOKA) [154] has for example extended the UML through stereotypes to form the Moka modeling language for the description of products and design processes. The Moka modeling language, however, contributes only little to product data integration as it does not represent application-specific model information. UML classes and activities are thereby extended by lightweight extensions which denote general product-related concepts such as assemblies, parts and attributes.

Similarly, the UML has also been extended for the conceptual modeling of mass-customizable products such as configurable personal computers [42]. Another example is MECHATRONIC UML [24] for the design of self-optimizing mechatronic systems [23], whereby UML component and statechart diagrams are extended to describe feedback controllers and the dynamics of physical systems.

A prominent example of a UML extension is the Systems Modeling Language (SysML) for the specification, analysis, design, verification and validation of a broad range of systems. SysML includes additional constructs for modeling system requirements, behavior, structure and parametrics [44]. SysML is suited for product modeling [11]. Peak et al. show how SysML supports simulation-based engineering design and analysis through SysML parametrics concepts which are applied to a mechanical example that integrates computer-aided design and engineering analysis (CAD/CAE) [129, 130]. SysML version 1.1 was released in 2008. Therefore, this modeling language cannot be considered

as mature and stable as UML which has already undergone major improvements since 1997 [89]. So changes in the new SysML modeling language are highly probable. Although SysML is already an extension of UML, it can itself also be extended for more specific domains. SysML has been extended for the simulation of mechatronic systems by integrating bond graphs [159] and the Modelica modeling language [132, 80].

SysML is a new modeling language which will probably undergo changes in the near future. However, once SysML has reached a mature and stable status, it will probably be better suited for multidisciplinary product data integration than the UML. Since SysML is based on UML, UML extensions supporting the representation of product information can eventually be reused to a large extent in SysML.

Current UML-based integration approaches are largely focused on software. So far, UML extensions often enable a domain-specific representation which is too abstract to be interpreted. No UML extensions for example currently exist to represent detailed application-specific models from mechanical engineering.

3.6 Summary

Central product models use generic modeling concepts in order to represent specific modeling concepts of various disciplines. Although models from different engineering disciplines are highly diverse, most models which are edited with current state-of-the-art software applications share common modeling concepts in order to support modular design. The capacity to easily exchange model components and to reuse model components across several models promotes flexibility and productivity in modeling. In order for engineers to easily recognize their modular-structured model information within a larger central product model, the central product model needs to be comprised of generic modeling concepts which can describe in detail modular components. Object-oriented modeling concepts are used to describe modular software. As they are generic, they can also be used beyond software modeling for the representation of specific modular model components of various engineering disciplines. The modeling concepts of a central product model require standardization as a central product model is intended to be used by many parties. The Unified Modeling Language (UML) is the de facto standard for object-oriented modeling and is widely used in software engineering. This thesis therefore investigates UML extensions in order to establish a central product model.

Chapter 4

UML profiles for geometric models

The geometric model of a product plays an important role in engineering design as it has an impact on many other product aspects. Apart from documentation and packaging studies, geometric models are for example used to drive the computer numerically controlled (CNC) machining of complex surfaces or to generate meshed models for structural and aerodynamic analyses. On the one hand, the UML-based product model needs to capture geometric information which is commonly shared in a multidisciplinary context. This includes for example important geometric characteristics such as volume, mass, center of gravity and moment of inertia. On the other hand, the UML-based product model needs to represent application-specific geometric modeling concepts in order to automatically translate the UML-based representation of geometric information into application-specific geometric models. Chapter 4 presents UML extensions for the UML-based representation of geometric information. Sections 4.1, 4.2 and 4.3 respectively present the mapping of CATIA-, SolidWorks- and VRML-specific geometric models into UML.

4.1 UML profile for CATIA-specific geometry

CATIA[1] is a top end geometry authoring application widely used in the engineering industry and was therefore chosen for this research work. In this Section, the mapping of parts, part parameters, part dependencies, assembly constraints, assembly models, user defined features and scripts into a UML-based product model is presented.

4.1.1 Parts

Modern CAD tools offer a multitude of features and operations to define geometry on the 2D level and to extrapolate it to 3D. According to the CATIA terminology, geometric data

[1] DASSAULT SYSTEMES, `http://www.3ds.com/products/catia/`

is structured according to the following hierarchy: 2D geometry is contained in sketches, 3D geometry in bodies or geometrical sets which in turn are contained in parts. A part represents the main modular model component which supports the reuse and exchange of geometric information across several models. An assembly of parts is called in CATIA a product.

If a part is to be updated, it would be a waste of time to update each single part in every product one at a time. For this reason, a CATIA part is a template, just as a class in object-oriented modeling. Each occurrence of a part in an assembly is an instance of a part. If a part is updated, all part instances are updated automatically. This object-oriented modeling paradigm plays a central role in CATIA and other CAD tools. It is thus intuitive to map a CATIA part into a UML class tagged with a «$catiaPart$» stereotype. Fig. 4.1 shows an example of a class corresponding to a part representing a rod. Part instances are translated into UML as class instances (Fig. 4.1 right). To enable the loading of an existing part during the automatic translation from UML into CATIA, the «$catiaPart$» stereotype has an optional *filePath* attribute of type String to indicate the file path to the existing part document.

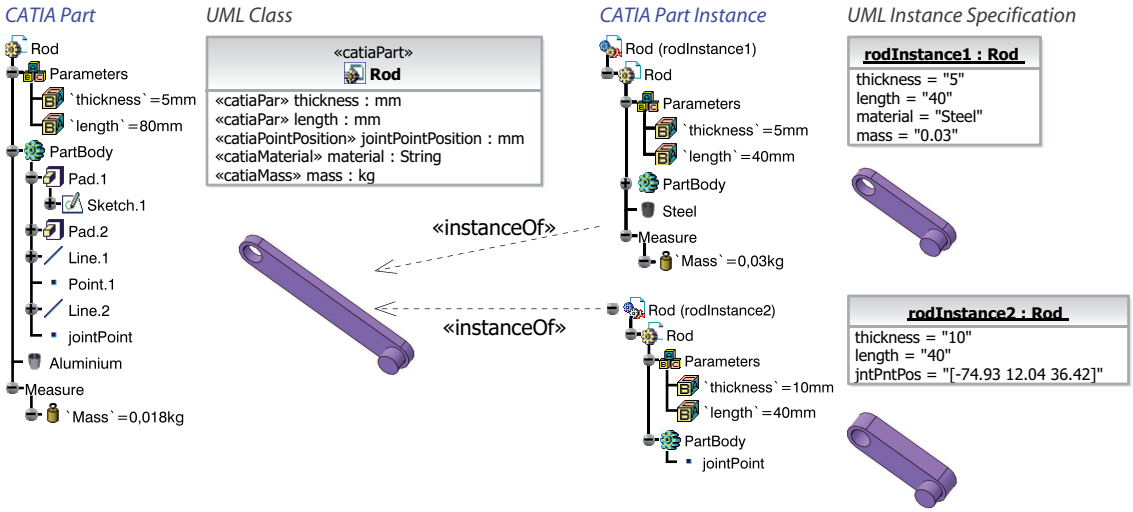

Figure 4.1: CATIA part and part instances with corresponding UML class and instance specifications

The elements owned by the rod part are displayed left in Fig. 4.1 in a typical CATIA tree structure composed of geometric elements. These geometric entities can be described as properties of the corresponding rod UML «$catiaPart$» class. Properties can for example be of type Body to describe this type of geometry container, or of type Point, Line or Plane for geometric elements. The mapping of geometric primitives such as points and lines into UML is represented in Sections 4.2 and 4.3.

4.1.2 Part parameters and measures

A comfortable method in CATIA to customize the geometry of a part without manipulating geometric primitives such as points and planes is through parameters. A part can be tailored for different configurations through the use of different parameter values. This approach is referred to as parametric design. The rod part has for example two parameters to tune its thickness and length (Fig. 4.1 left). These parameters are common to all rod part instances. Examples of part instances with different parameter values are shown in Fig. 4.1 right. Parameter values set in the part definition are default values for all part instances, but part instances can individually overwrite these values. Parameters of CATIA parts are mapped as UML properties of the corresponding UML «$catiaPart$» class and tagged with a «$catiaPar$» stereotype. The datatype of the property corresponds to the parameter unit, such as mm in the example above. The referenced datatype can be either predefined in a package containing all the SI value types or introduced as needed into the UML model. A similar but unique part parameter is used to describe the material applied on the part. The material value can, like normal parameters, vary between part instances (Fig. 4.1). Because of its unique nature, the material Property is tagged with a «$catiaMaterial$» stereotype (Fig. 4.1 left).

CATIA offers the possibility to measure characteristics of geometric elements or of complete parts. Measures such as the mass, the centre of gravity or the inertia matrix of a part are valuable part properties which often have an impact on other engineering domains. These values are also hard to determine manually when the geometry is complex. Important part measures are translated as properties of the related class with appropriate stereotypes such as «$catiaMass$», «$catiaCG$» or «$catiaInertia$». The example in Fig. 4.1 displays the mass measure in the tree structure of the rod part and the related «$catiaMass$» property in the rod class. The names of the properties can be arbitrary as the correlation with the CATIA measure is defined by the stereotype. Similarly the measurement of point coordinates is a frequent operation. In UML, this measure is translated as a UML property tagged with a «$catiaPointPosition$» stereotype. To enable the UML property to have a different name from the measured point, the name of the point to be measured is saved in an attribute of the respective «$catiaPointPosition$» stereotype.

Measure and parameter values of CATIA part instances are represented in UML as *literal string* values in the *slots* of the related UML instance specifications. The UML part instance rodInstance1 in Fig. 4.1 for example has a slot with a *defining feature* corresponding to the thickness property of the *«catiaPart»* rod class and with a literal string value of “5”.

4.1.3 Dependencies between parts

Dependencies between parts are frequent. The dependent part is often referred to as the child part and the referenced part as the parent part. The child part is then updated when the parent part changes. For instance, the lowermost form of the slider part (Fig. 4.2 left) is dependent on the rail profile form of the base part represented by a sketch (Fig. 4.2 right). There are two methods for a child part to reference an element of a parent part. One is direct but does not enable a later replacement of the parent part with a similar part. In the previous example, this would result in the slider part directly referencing the Sketch.2 element of the base part. In software terms, this would be similar to hard coding, which is considered an anti-pattern. The downside of this direct approach is that if the base part is replaced with a similar part, the previously specified dependency needs to be redefined.

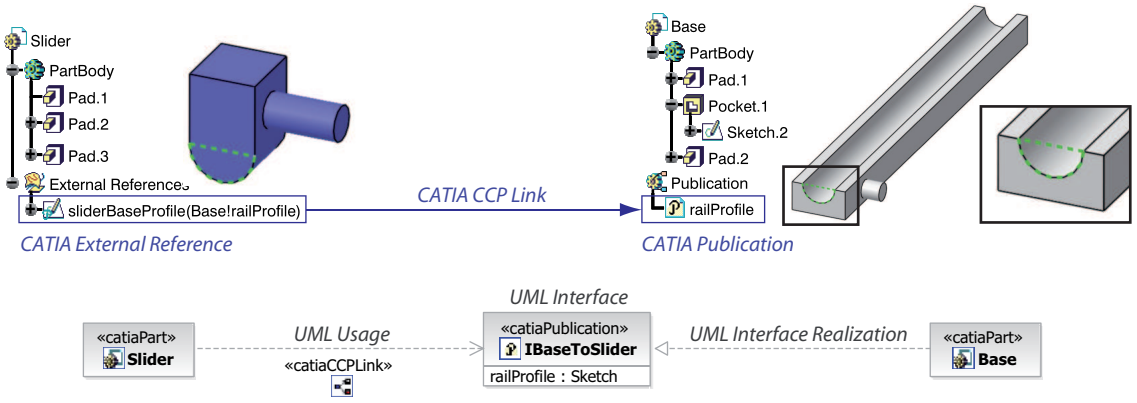

Figure 4.2: Top: CCP link between the slider and the base part via a published sketch element. Bottom: Dependency described in UML through related «$catiaPart$» classes requiring and providing a «$catiaPublication$» interface

The other more flexible referencing method in CATIA consists of using an intermediary interface. The parent part explicitly declares the elements that are to be made easily available to other parts. These are called the published elements and are owned by the parent part. A published element is linked with the geometric entity it is representing. The published element can also have a more descriptive name than the geometric element it is standing for, such as railProfile instead of Sketch.2 (Fig. 4.2 right). The child part then does not directly reference the geometric entity in the parent part but its representative, namely the corresponding published element of the part. This allows to decouple the reference from the concrete referenced geometric entity, allowing the reference to stay the same although the referenced geometric entity may change. The parent part can be swapped for another one while the previously defined dependency will persist under the

condition that the new parent part offers the same published elements. Published elements are identical if they have the same name and are of the same type. The use of publications makes the exchange of geometric entities more transparent and enables a more modular part architecture. The base part for example publishes the sketch of the rail track under a publication called railProfile and the slider references this publication (Fig. 4.2).

The concept of a publication is similar in object-oriented software design to an interface describing a service required by one class and offered by another. It is often a recommended practice in software design to define a dependency between two software classes via an interface-typed reference that can stay the same while the implementing class can easily be swapped. Due to the similarity between the concept of a publication in CATIA and of an interface in object-oriented software programming, the publications list of a CATIA part is translated in UML into a UML interface tagged with a «$catiaPublication$» stereotype. And the class corresponding to the CATIA part has an interface realization relationship with the «$catiaPublication$» interface. The published elements are translated into attributes of the «$catiaPublication$» interface with corresponding names and types. In the example of Fig. 4.2, the «$catiaPart$» base class realizes the «$catiaPublication$» IBaseToSlider interface, owning the published element railProfile of type Sketch.

The dependency of a part on a geometric entity of another part independent of any assembly context is called in CATIA a Cut Copy and Paste (CCP) link. As the use of publications is recommended, the dependency of a child part is not directed at the parent part but at the published geometric entities. The CCP link dependency is described in UML as a UML usage dependency between the UML class representing the child part and the UML interface corresponding to the published elements list. The usage dependency is tagged with a «$catiaCCPLink$» stereotype. The «$catiaPart$» Slider class of Fig. 4.2 for example has a «$catiaCCPLink$» usage dependency on the «$catiaPublication$» IBaseToSlider interface. The railProfile sketch element will need to be provided by an implementing class, in this case the «$catiaPart$» Base class.

4.1.4 Products

It is common to decompose the entire geometry of a product into several parts to facilitate the reuse of single parts in other projects. An assembly of part instances is called in CATIA a product. An example of a product composed of several assembled part instances forming a slider-crank mechanism can be found in Fig. 4.3. The sliderMotion product is composed of skeleton, base, crank, rod and slider part instances. All part instances are visible except the skeleton instance which only consists of a 2D sketch describing the main kinematic features of the slider mechanism (Fig. 4.7).

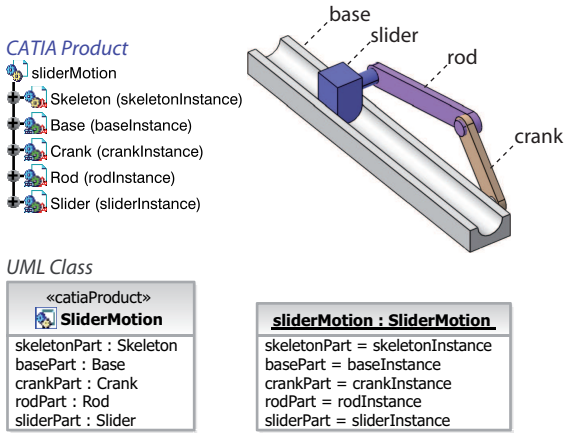

Figure 4.3: CATIA product and related UML class and instance

As large assemblies can be composed of smaller assemblies, the decomposition of the geometry into products can also occur. So a product is, just as a part, also a template that can be instantiated. A product standing for a large assembly can hence contain product instances representing smaller assemblies. Due to the similarity of a product to a part in respect to their common template nature, a CATIA product is translated like a CATIA part in UML into a UML class, but tagged with a «$catiaProduct$» stereotype. Products can, just like parts, have parameters to tune their geometry, have measures such as their mass and also publish their geometric entities. So «$catiaProduct$» classes can have properties tagged with the same stereotypes previously presented for properties of «$catiaPart$» classes and also implement «*catiaPublication*» interfaces. To support an executable translation of UML into CATIA, the «$catiaProduct$» stereotype also has the optional *filePath* attribute to refer to an existing product. Furthermore, the topmost product in the product hierarchy to be considered for the UML to CATIA conversion is tagged with a «$catiaRootProduct$» instead of a casual «$catiaProduct$» stereotype.

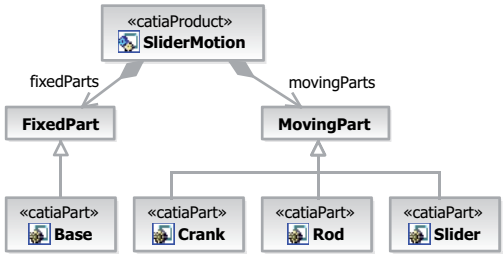

Figure 4.4: Possible introduction of non CATIA-specific classes for a better classification

The part or product instances owned by a product are described in UML as properties of the «$catiaProduct$» class. The properties are typed with corresponding «$catiaPart$» or «$catiaProduct$» classes. The sliderMotion class for example has properties whose

types are respectively equal to the «$catiaPart$» skeleton, base, crank, rod and slider classes (Fig. 4.3 bottom left). But the mapping between the instances owned by the CATIA product and the properties owned by the «$catiaProduct$» class does not need to be one to one. New classes can be introduced in UML that do not have their equivalent in CATIA. This enables a better higher level classification of the «$catiaPart$» classes. The parts of the sliderMotion product can for example be separated into fixed and moving parts (Fig. 4.4). Generalization relationships would then exist between the «$catiaPart$» classes and the non CATIA-specific classes. The sliderMotion «$catiaProduct$» class instance would then reference its owned part instances via its fixedParts and movingParts slots.

4.1.5 Assembly constraints

CATIA assembly constraints ensure the correct positioning and orientation of part or product instances relative to each other within an assembly. An assembly constraint therefore references the involved geometric entities and according to its type restricts the relative movement of these entities. The revolute axes of the rod and slider part instances for example need to coincide (Fig. 4.6), so that if one part is displaced, the other part will be displaced accordingly. Assembly constraints are owned in CATIA by the product. As constraints need to reference specific geometric entities within parts, it is recommended that they reference the published representatives of concerned entities. Through the use of published elements playing an intermediary role, the constraint can stay the same while the concerned parts or features are replaced with similar ones.

The published elements of a CATIA part are contained in only one unnamed list called the publications list whereas a UML class can implement several UML interfaces. In the first case, the origin or type of a dependency cannot be recognized as all the published elements are stored in the same publications list independent of their role. In the latter case, each UML interface can depict for higher modularity and clarity a certain type of service between a realizing and a client class. The limitation of mapping all published elements into one UML interface does not apply. The published elements translated into UML can be separated into several interfaces according to their roles, allowing a better overview of the different publishing intentions. It is for example the case with the base part in Fig. 4.5, which provides the railProfile sketch marked as a green dashed line for the slider according to the IBaseToSlider interface and other geometric entities for assembly colored green and corresponding to the IBaseForAssembly interface.

A CATIA assembly constraint is translated into a UML constraint. CATIA constraints are owned by a product, so the UML constraints are owned by the related «$catiaProduct$»

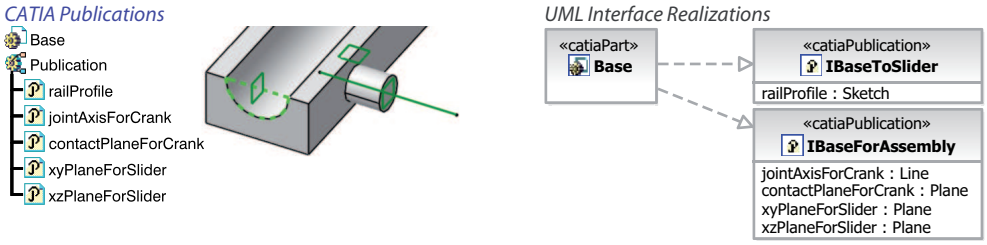

Figure 4.5: Published elements of the base part mapped into different UML interfaces

class. According to the type of the assembly constraint, the UML constraint is tagged with a specific stereotype such as «$catiaFix$» or «$catiaCoincidence$». If necessary, the specific stereotype owns attributes for a complete description of the constraint. A «$catiaAngle$» stereotype for example owns attributes to specify an angle value and an angle sector.

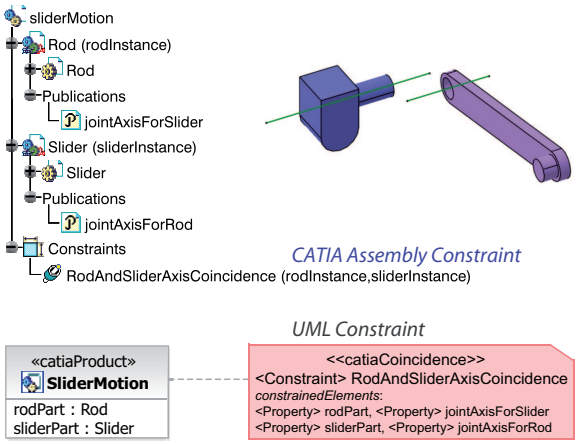

Figure 4.6: CATIA coincidence assembly constraint and respective UML constraint

The *constrainedElement* attribute of the UML constraint needs to refer to the elements corresponding to the geometric part entities confined by the CATIA constraint. The UML constraint can therefore refer either to the instance slots or to the «$catiaProduct$» class properties representing the constrained geometric elements. In the first case, the constraint only applies to specific part instances. In the latter, the constraint applies to all «$catiaProduct$» class instances but can only be successfully resolved if the properties representing the constrained parts have a *multiplicity* equal to one.

In the first case, the constraint refers to the product instance concerned and to the instance slots whose defining features correspond to the published geometric elements described as «$catiaPublication$» interface properties. In the second case, the constraint references sets of two properties, one representing the involved part and one the involved geometric entity. The UML coincidence constraint of Fig. 4.6 for example references the

rodPart and sliderPart properties of the «$catiaProduct$» sliderMotion class as well as the properties representing the axes owned by the respective «$catiaPublication$» interfaces.

4.1.6 Dependencies between part instances

CATIA supports the design of a part in relation to already existing parts in an assembly. This design method is referred to as relational design or as design in context. The resulting part dependencies, called import links, are only active in a certain assembly context, unlike CCP links which define part dependencies independent of any assembly context.

CATIA import links are often used in combination with a so-called skeleton or adapter model which significantly reduces the overall number of part dependencies. The skeleton model describes the geometric entities upon which the rest of the assembly will be built and enables to easily configure the entire assembly by only adapting the skeleton model. The skeleton model usually only contains basic geometric entities such as points, lines, planes and sketches to indicate the positioning or the dimensions of the concrete parts making up the assembly. The use of a skeleton model for the authoring of geometry is in its principle similar to the use of the template method design pattern in software engineering [47] which describes a basic frame, also called “skeleton”, in which complex software components can fit in.

In the case of the sliderMotion product, the skeleton model, represented by a skeleton part, describes the dimensions that apply to the neighboring parts such as the base, the crank, the rod and the slider (Fig. 4.7 left). The use of publications is again recommended as in every case in which a part, product or constraint references a geometric entity. So the skeleton part publishes the parameters such as crankLength or rodLength.

The published entities are partitioned into several UML interfaces according to the interaction the skeleton «$catiaPart$» class has with each neighboring «$catiaPart$» class (Fig. 4.7 right). The jointRadius parameter, which describes the radius of every revolute joint cylinder between the bodies, is present in each interface, as every part depends on this value. The neighboring parts declare their dependency on the skeleton part by having a UML usage dependency on the «$catiaPublication$» interfaces tagged with a «$catiaImportLink$» stereotype, just as in the case of CCP links.

As an import link dependency is only present in the context of an assembly, it is described in UML as a UML assembly connector between parts of the composite structure of a «$catiaProduct$» class (Fig. 4.8). The “ball-and-socket”notation used in Fig. 4.8 represents the parts offering and requiring interfaces and is available if the parts have as classifier a UML component instead of a class. A UML component is a subtype of a class and as a consequence has the same properties as a UML class. So the mapping of a CATIA

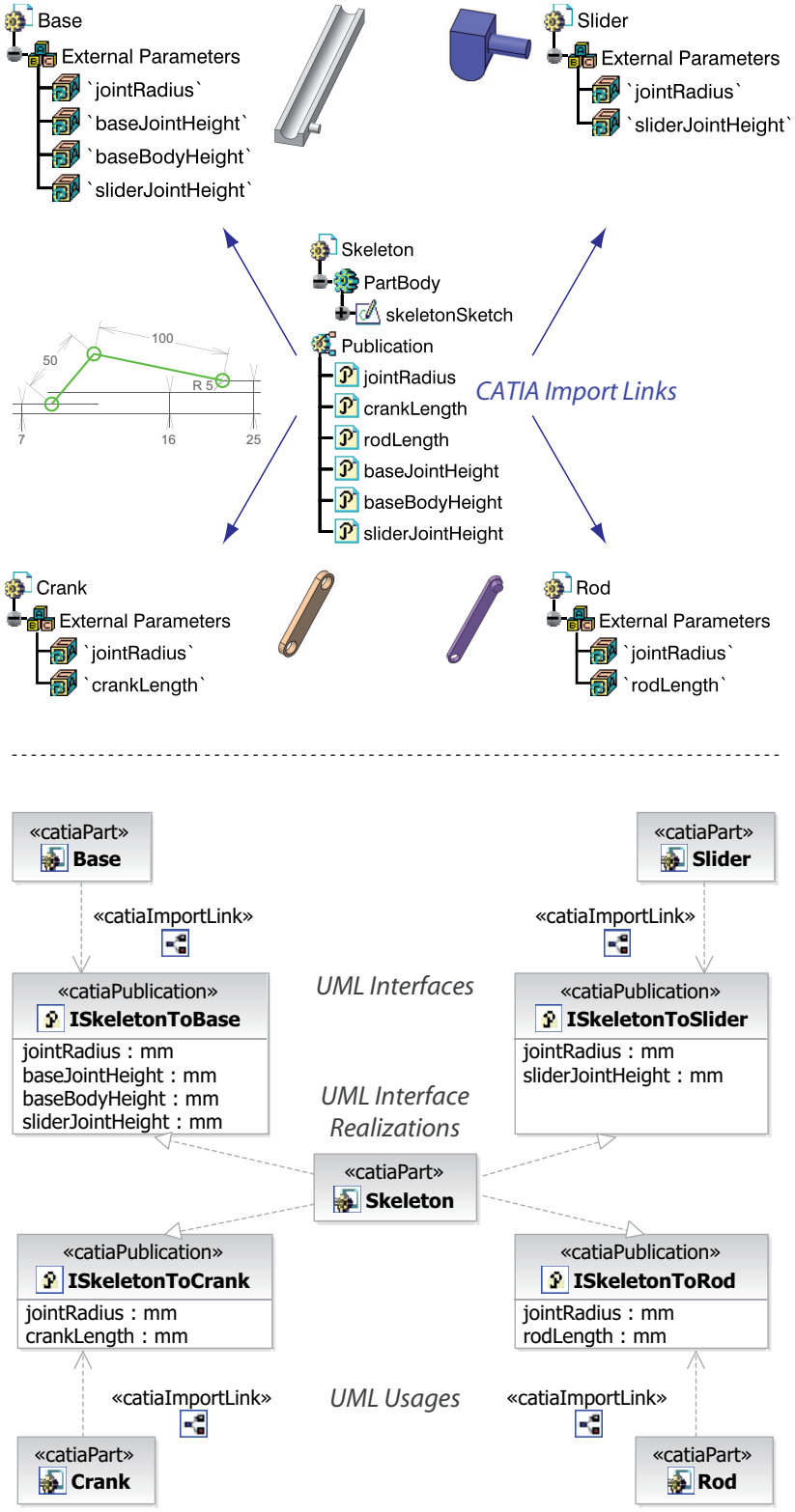

Figure 4.7: Top: CATIA import links between the skeleton and its neighboring parts. Bottom: Corresponding UML «*catiaImportLink*» interface usages between related «*catiaPart*» classes

product or part into a UML class can be replaced with a mapping into a UML component. The stereotypes previously applied on classes can also be applied on components. The composite structure diagram of the sliderMotion component (Fig. 4.8) shows the wiring of its owned parts. Through the “ball-and-socket”notation it is for example easily visible that any crank part instance playing the role of a crankPart needs to refer to the geometric entities of a skeleton part instance playing the role of a skeletonPart in the context of a sliderMotion product instance.

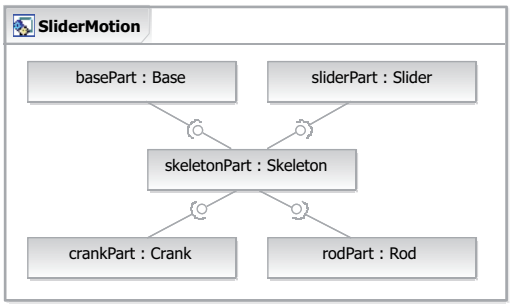

Figure 4.8: CATIA import links displayed as UML connectors in the composite structure diagram of the related «$catiaProduct$» class

4.1.7 PowerCopies

CATIA PowerCopies enable the reusability of a design intent and are very similar to user-defined features. Common geometric operations are for example defined as PowerCopies. PowerCopies are comparable to operations with input arguments and an outcoming result. They represent knowledge templates which can be instantiated in different geometrical contexts. However, a PowerCopy is not defined by specifying the single steps to reach a target feature. Instead, it is defined according to an already existing target feature. CATIA detects all geometric entities such as points, lines and faces, and all non-geometric entities such as parameters and formulas which compose a desired target feature. The user chooses from this set the entities which are to be variable. These then represent the input arguments of the Powercopy which is either saved in a part or in a catalog. A PowerCopy can be reused in another context, such as in another part, to create a similar target feature however based on different input arguments. The features inserted by the application of PowerCopies are not necessarily of geometric nature. They can also include non-geometric aspects such as checks and rules.

The example in Fig. 4.9 shows the definition of the CreateFillet PowerCopy. It creates an EdgeFillet feature and a formula based on two faces and a radius value. The target EdgeFillet feature, displayed in purple in the 3D model, is the result of applying a fillet

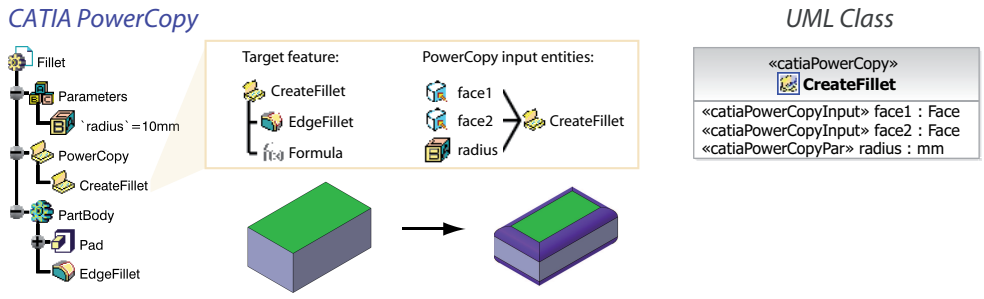

Figure 4.9: CATIA PowerCopy and related UML class

with a radius of 10mm on both green box faces. The application of the CreateFillet PowerCopy on another part is shown in Fig. 4.10. The PowerCopy is instantiated with two crank part faces colored green and a radius value of 2mm as input arguments. The resulting crank fillets are colored purple. The result of the PowerCopy application, which is synonymous with its instantiation, is displayed in the design tree of the crank part as edgeFilletInstance feature.

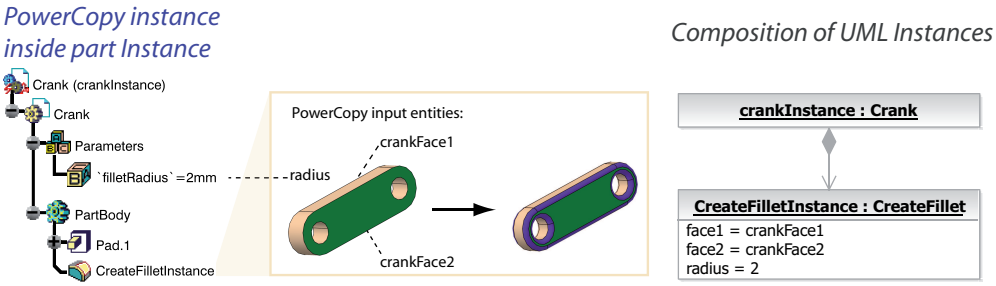

Figure 4.10: CATIA PowerCopy instance and related UML instance

As PowerCopies represent knowledge templates, they are represented in UML as classes. The PowerCopy-specific classes are tagged with a «$catiaPowerCopy$» stereotype. PowerCopy input parameters are described as UML properties of the «$catiaPowerCopy$» class and are tagged with a «$catiaPowerCopyPar$» stereotype. Accordingly, PowerCopy input entities are described as UML properties and are tagged with a «$catiaPowerCopyInput$» stereotype. The application of PowerCopies are correspondingly described as UML instances of «$catiaPowerCopy$» classes.

The CreateFillet PowerCopy of Fig. 4.9 is for example described in UML as a «$catiaPowerCopy$» class with its face-related properties tagged with a «$catiaPowerCopyInput$» and its radius property tagged with a «$catiaPowerCopyPar$» stereotype. The CreateFilletInstance PowerCopy instance for example references both crank faces as input arguments and sets the radius value to 2mm (Fig. 4.10). The PowerCopy instance is inserted into the crankInstance part. The composition relationship between the instances

is mapped one-to-one into UML, so the related UML crank part instance is composed of the UML PowerCopy instance.

CATIA user defined features are knowledge templates very similar to PowerCopies. They are mapped into UML just as PowerCopies. The main difference lies in the black box character of user defined features. The result of a PowerCopy instantiation is visible in the CATIA design tree. It might for example show the entities which have been added through the application of a PowerCopy. On the other hand, the features added by the user defined feature instantiation are not visible in the design tree. According to the previous example, a user defined feature instance corresponding to the PowerCopy instance would not represent the fillet feature in the design tree. The application of a user defined feature is only represented in the design tree through a reference to a user defined feature instance. As a consequence, a user defined feature enables to insert design know-how without disclosing the details of a geometric operation.

4.1.8 Scripts

CATIA supports several programming languages, such as Visual Basic Script (VBS), Visual Basic Application (VBA), Java and C++, for the automation of routine geometry editing tasks. A CATIA-specific VBS/VBA program is also called a script or macro. Routine geometry steps are programmed in a script and can be executed and reused in different contexts. The CATIA script in Fig. 4.11 for example creates a pocket feature based on a sketch and a depth value. Its execution is shown exemplarily in Fig. 4.12.

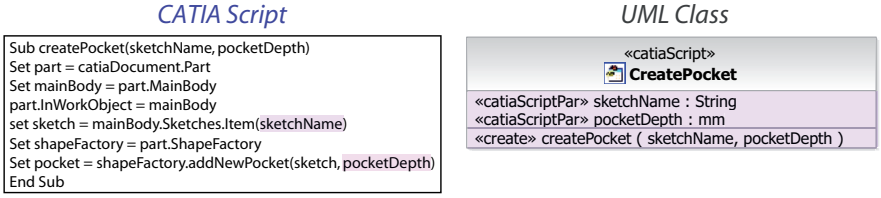

Figure 4.11: CATIA script and corresponding UML class

As scripts represent a program, they can be translated into UML operations. The result of a script activation is a feature which is represented in UML as an instance. To indicate that a UML instance is the result of a UML operation, the classifier of the instance can own the respective operation and declare it as a constructor operation with a single return result of the type of the owning class. A constructor operation is typically tagged with a «*create*» stereotype (Fig. 4.11).

Although the mapping of a CATIA script into a UML operation is feasible, a less modeling-intensive and simpler UML representation of a CATIA script is possible. A

script represents a template which can be executed, in other words instantiated. A script can thus be described in UML through a UML class and a script activation through a UML instance. The application of the «$catiaScript$» and «$catiaScriptPar$» stereotypes respectively on the UML class and UML properties is sufficient to unambiguously describe in UML a CATIA script and its parameters. The activation of a script is then described through a corresponding UML instance.

The location of the CATIA script is described in the *catiaScriptPath* attribute of the «$catiaScript$» stereotype. The execution order of scripts may be important since some may be based on the result of a previous script execution. In this case, the execution priority of a script is set through the *order* attribute of the «$catiaScript$» stereotype.

A script is executed within a part or a product instance. As a result, the part or product instance owns the new features created by the script activation. The composition relationship between the part or product instance and the result of the script activation are translated into UML as a composition relationship between the related UML instances.

The execution of the createPocket script of Fig. 4.11 in a slider part instance is for example displayed in Fig. 4.12. In this case, the script activation only adds a pocket feature to the slider part instance. As a result of the script execution, the design tree shows the new pocket feature named pocketInstance and the geometric model displays a hole in the slider part. Similar to the composition of CATIA features visible in the design tree, the corresponding UML slider instance owns the pocket instance which represents the result of the script activation. The input values for the activation of the createPocket script are stored in the slots of the pocket instance whose classifier is the «$catiaScript$» CreatePocket class.

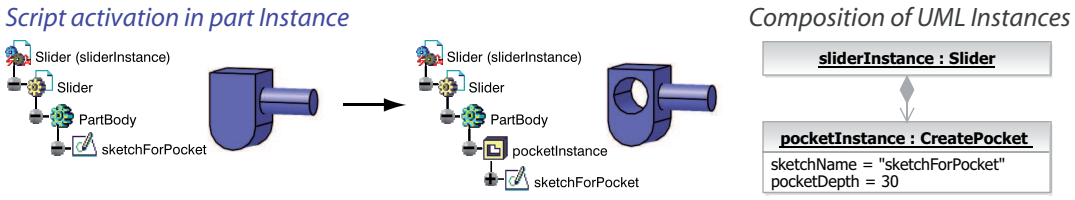

Figure 4.12: Script activation inside a part instance and corresponding UML instances

4.2 UML profile for SolidWorks-specific geometry

SolidWorks[2] is a popular 3D mechanical computer-aided design (CAD) application which shares common modeling concepts with CATIA but also has some of its own. This Sec-

[2]DASSAULT SYSTEMES, `http://www.solidworks.com/`

tion first presents the mapping of SolidWorks-specific assemblies into UML similar to the mapping of CATIA-specific products into UML. The Section then shows the mapping of detailed geometric entities such as planes and axes into UML, which was not addressed in the last Section. Finally, the Section shows the UML-based representation of SolidWorks-specific assembly constraints which are defined differently than in CATIA.

4.2.1 Assemblies

The definition of geometry in SolidWorks is very similar to that in CATIA. 2D geometry is defined in sketches, which are extrapolated to 3D geometry, which itself is saved in part or assembly documents. Both parts and assemblies are templates which can be inserted through instantiation into other assemblies. Figure 4.13 shows the Solidworks assembly model of the slider-crank mechanism and its corresponding design tree. The assembly contains part instances as well as geometric entities such as planes and axes.

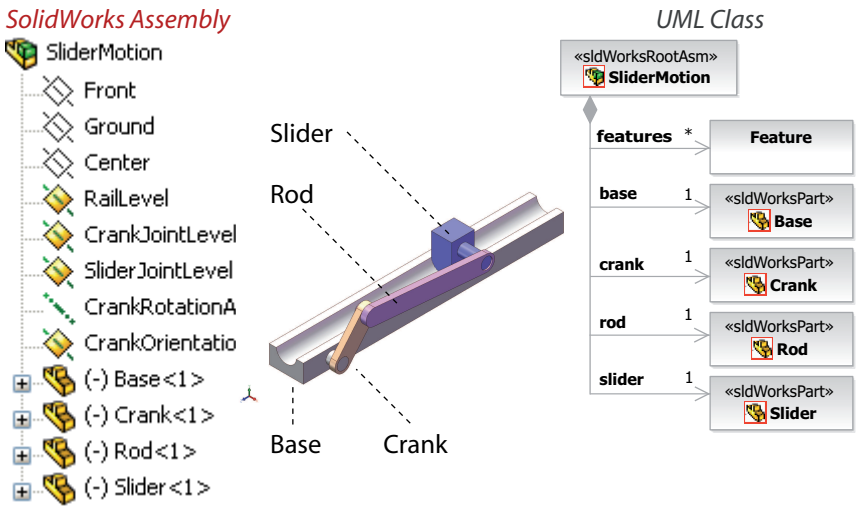

Figure 4.13: SolidWorks assembly model of the slider-crank mechanism

The mapping of SolidWorks parts and assemblies into UML is identical to the mapping of CATIA parts (Subsection 4.1.1) and products (Subsection 4.1.4) into UML with the only difference being the stereotype names. The SolidWorks assembly of Fig. 4.13 named sliderMotion is for example translated into a UML class tagged with a «$sldWorksRootAsm$» stereotype. The owned parts, such as the base, the crank, the rod and the slider, are described as UML classes tagged with a «$sldWorksPart$» stereotype. Assemblies are normally depicted in UML with a «$sldWorksAsm$» stereotype but the highest assembly in the model hierarchy is tagged with a «$sldWorksRootAsm$» stereotype to reflect its unique role. The resulting class diagram showing the composition between the assembly and its owned parts is represented in Fig. 4.13 right.

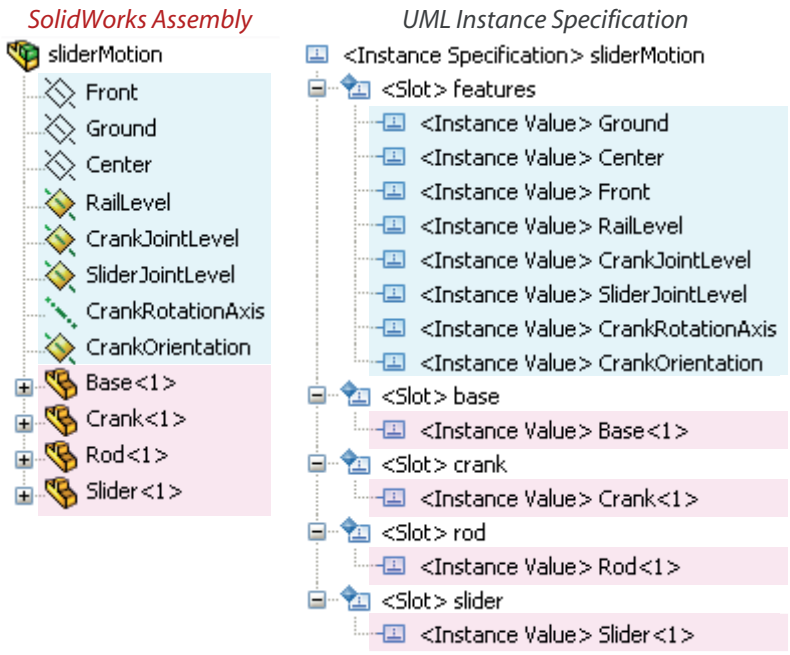

Figure 4.14: UML instance of the slider-crank mechanism SolidWorks assembly model

The tree structure of the UML instance corresponding to the SolidWorks assembly is shown in Fig. 4.14. The slots of the sliderMotion instance correspond to the properties of its «$sldWorksRootAsm$» SliderMotion class. The slots of the sliderMotion instance contain instance values which refer to instance specifications. The features slot for example contains instance values referring to the instances of the geometric entities such as planes and axes. Each other slot refers to an owned part instance. An ordering of geometric entities and parts in another set of UML properties according to other criteria is possible. The current example closely mirrors the SolidWorks composition structure in UML.

Each assembly has three reference planes according to which its owned geometric entities are positioned. In the example of the slider-crank mechanism, the planes are named Front, Ground and Center. To differentiate the reference planes from casual planes, the corresponding UML instances are tagged with a «$sldWorksRootPlane$» stereotype.

4.2.2 Geometric entities

A SolidWorks assembly can directly contain on the same composition level detailed geometric entities in addition to part or assembly instances. The sliderMotion assembly in Fig. 4.13 is for example composed of several planes such as RailLevel or CrankJointLevel. CATIA assembly models are on the other hand only composed of part or product instances and only indirectly contain geometric entities through their owned part instances. Geo-

metric entities play a more important role in SolidWorks than in CATIA. The SolidWorks geometry is therefore often changed through the modification of detailed geometric entities. It is hence necessary to represent detailed geometric entities in UML.

Geometric entities are often defined according to different sets of argument types as well as based on other geometric entities. An axis can for example be defined either by being perpendicular to a plane and passing through a point or by going through two points. The type of a geometric entity specifies the invariant set of possible defining arguments.

Geometric entity types represent a class of similar geometric entities and are therefore described in UML as classes. Each geometric entity type is defined according to specific arguments which are described as properties of their respective UML entity type-specific class. To avoid a redundant definition of invariant geometric entity type-specific UML classes in every UML model, they are predefined only once in the Solidworks profile. Specific UML classes corresponding to geometric entity types, such as Plane and Axis, have therefore been defined in the SolidWorks profile.

A geometric entity instance, such as the RailLevel plane of the sliderMotion assembly in Fig. 4.13, is for example represented in UML by an instance whose classifier is the predefined Plane class of the SolidWorks profile. The reference to predefined domain-specific classes does not hinder a simultaneous multiple-domain interpretation as instances can have several classifiers. As an example, a SolidWorks-specific UML plane instance which additionally needs to represent a CATIA-specific plane can have as second classifier the predefined Plane class of the CATIA profile.

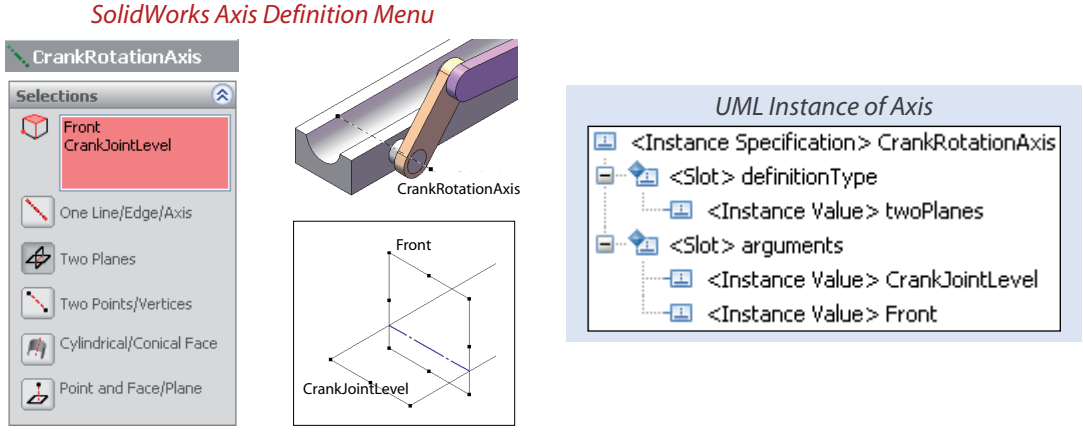

Figure 4.15: Axis defined by two planes and corresponding UML instance

The SolidWorks assembly model of the slider-crank mechanism in Fig. 4.13 contains geometric entities to establish a skeleton model upon which the owned parts can be positioned. The crank part must for example rotate around an axis common to the base part. The axis is defined as CrankRotationAxis in the sliderMotion assembly model (Figure

4.15). Both the base and crank parts are positioned according to it. The axis is defined as the intersection of two planes. As a consequence, the arguments to define the axis refer to two planes, namely the Front and CrankJointLevel planes. Figure 4.15 shows the SolidWorks menu to define the axis based on different arguments. The different definition types are described in the SolidWorks profile as enumeration literals. The UML instance describing the CrankRotationAxis is of type Axis and refers through slots to the same information as in the SolidWorks menu. The first slot specifies the definition type by containing an instance value whose instance attribute is equal to the “twoPlanes”enumeration literal. The second slot specifies the input arguments through instance values referring to the CrankJointLevel and Front plane instances.

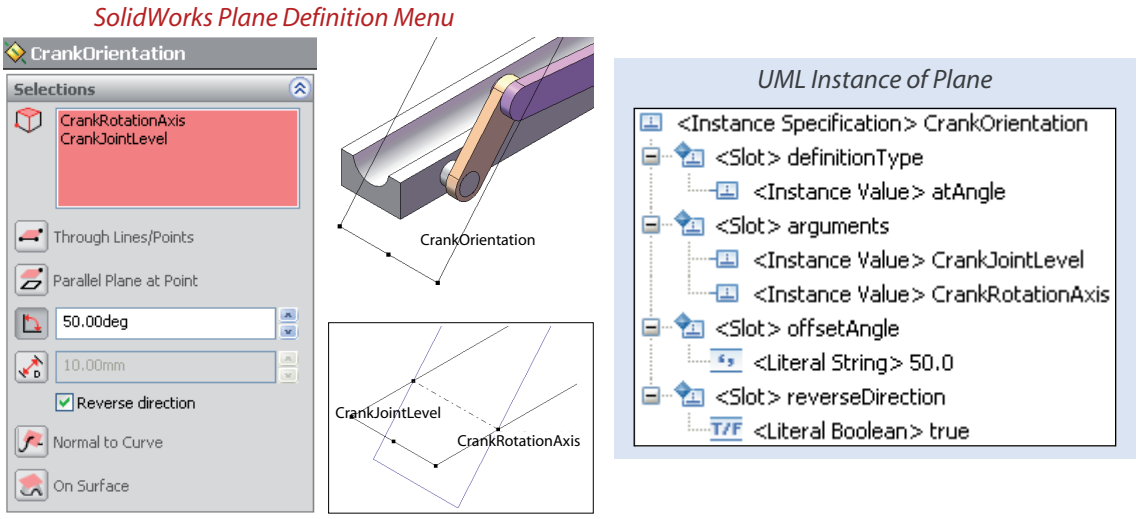

Figure 4.16: Plane defined by its angle to another plane and corresponding UML instance

Another example of a geometric entity belonging to the slider-crank mechanism assembly is the CrankOrientation plane displayed in Fig. 4.16 which defines the orientation of the crank. It is, just like the axis, also described according to a definition type, in this case “at angle” with a value of 50.00deg and relative to two arguments, namely the CrankRotation axis and the CrankJointLevel plane.

In contrast to part-related modeling concepts, geometric entities were represented in UML without stereotypes. A geometric part is a modular model component. As a result, the semantics of part-related modeling concepts corresponded to the semantics of generic UML modeling concepts with an applied stereotype. In this case, the combination of a stereotype with a UML modeling element formed a meaningful information unit. On the other hand, geometric entities do not form encapsulated modular entities but are often only meaningfully defined based on other geometric entities. As an example, the definition of an axis may require two points. While geometric entities can be mapped

one-to-one into corresponding UML elements, the application of stereotypes would not add any useful semantics since a geometric entity is only meaningfully represented in UML in combination with its dependent geometric entities.

4.2.3 Mates

Assembly constraints position the geometric elements relative to each other and are called in SolidWorks mates. The type of a SolidWorks mate is not visible in the design tree but only recognizable by opening its definition menu. CATIA on the other hand directly displays the type of constraint in the design tree through an icon. Due to this difference, SolidWorks mates are mapped slightly differently than CATIA assembly constraints into UML. A SolidWorks mate is described in UML as a constraint tagged with a non type-specific «$sldWorksMate$» stereotype. The arguments required to define a SolidWorks mate in UML are specified through the attributes of the UML constraint and of its applied «$sldWorksMate$» stereotype.

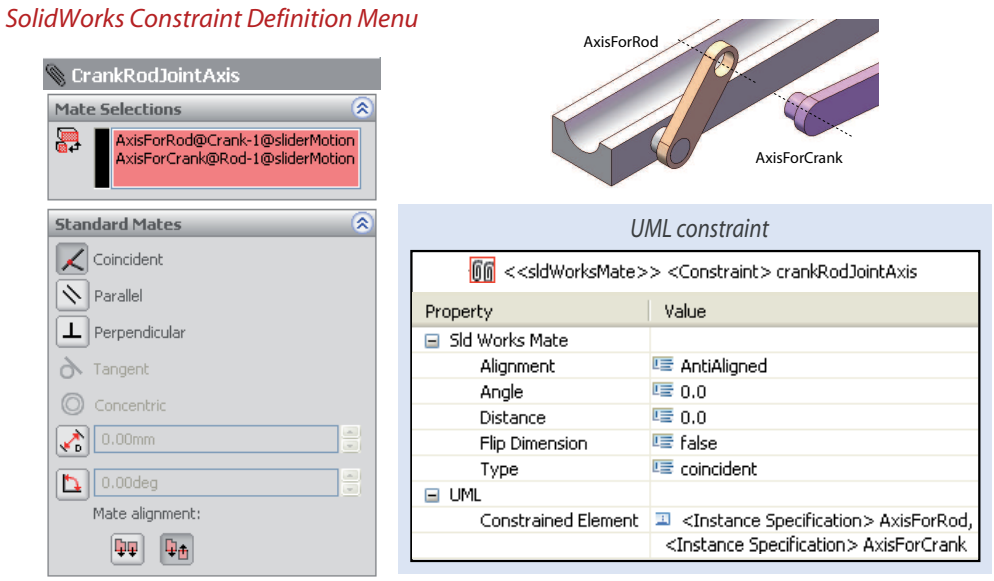

Figure 4.17: Coincidence constraint between two axes and corresponding UML constraint

A set of arguments and options depending on the mate type is required for the full definition of a SolidWorks mate. The mate-specific options such as type, distance, angle, flipDimension and alignment are described in UML through the properties of the «$sldWorksMate$» stereotype (Fig. 4.17). The alignment option of the SolidWorks coincidence mate is a supplementary argument to unambiguously define the orientation of the constrained geometric entities before mating. The flipDimension option enables to swap the direction in case of a distance mate. The geometric elements constrained by the mate

are described in UML as instances. The UML constraint, corresponding to the SolidWorks mate, references the constrained geometric elements through its constrainedElement attribute.

In the case of the slider-crank mechanism, the joint axis of the crank and of the rod for example need to coincide (Fig. 4.17 top right). A SolidWorks mate of type Coincidence having both axes as arguments is therefore defined. The alignment option is set to AntiAligned. As shown in Fig. 4.17, the mate type is not recognizable in the design tree of the assembly. The corresponding UML constraint is displayed in Fig. 4.17 bottom right. In case of distance and angle mates, distance and angle values are required.

4.3 UML profile for VRML-specific geometry

The Virtual Reality Modeling Language (VRML) is a file format capable of representing static and animated 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies and images. VRML browsers, as well as authoring tools for the creation of VRML files, are widely available for many different platforms.

The second version of the specification, known as VRML97 or VRML 2.0 [165], has been accepted as a standard format by the International Organization for Standardization (ISO) and the Web3D Consortium[3] was formed to ensure its development. VRML has been superseded by Extensible 3D (X3D) [166] which is a new ISO standard in an XML-based format. However, VRML is still very widespread and many 3D modeling programs support an automatic translation into VRML.

Subsection 4.3.1 presents the basic VRML file structure. Subsection 4.3.2 presents the mechanism used for the translation of detailed VRML-specific information into a UML model. Subsection 4.3.3 shows VRML files displaying less detail and thus less granularity as well as another mapping technique for the translation. Subsection 4.3.4 presents the UML-based linking of CATIA and VRML data to enable an automatic translation of CATIA into VRML via the UML-based product model.

4.3.1 File structure

A VRML file is defined in a text file with the .wrl ending for World. An example of a VRML plaintext file and corresponding geometry rendered by a VRML browser is displayed in Fig. 4.18. A VRML file essentially consists of a scene graph and an event routing. The scene graph hierarchically contains nodes which describe audio-visual ob-

[3]Web3D Consortium, `http://www.web3d.org/`

jects and their properties. The event routing is a mechanism to process events generated by nodes so that the scene graph can change dynamically. Additionally, a VRML file can contain prototypes which allow a set of VRML node types to be extended by the user. Their implementation is then browser-dependent.

The slider VRML file in Fig. 4.18 only contains a scene graph and no event routing nor prototypes. The file begins with a comment highlighted in dark grey which acts as a header to support an easier identification of the file type. Bindable nodes have an enumeration of properties whereby only one is active at any time. In the slider example, the bindable nodes Background and Viewpoint are highlighted in blue. If for example several viewpoints were defined, only one viewpoint could be active.

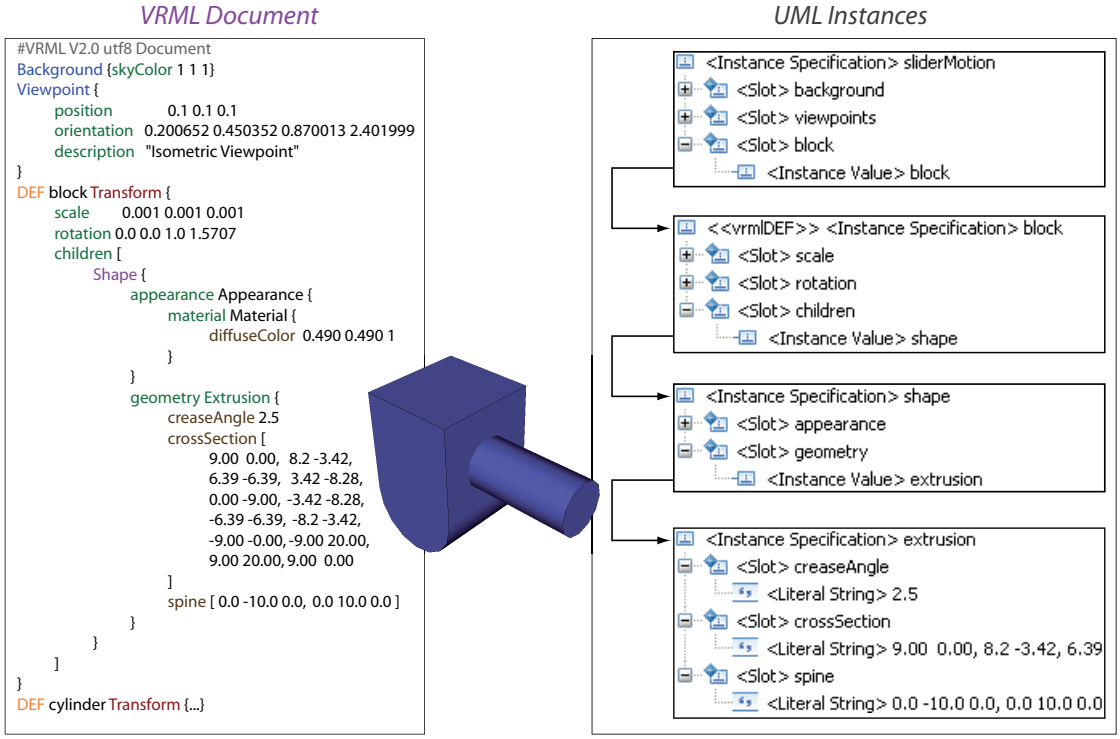

Figure 4.18: VRML-rendered geometry of the slider with corresponding sample VRML text file and UML instances

The scene graph consists in the slider example of two grouping nodes of type Transform. Their names are declared after the keyword DEF and are respectively “block” and “cylinder”. A grouping node defines a coordinate space for its child nodes. The Transform grouping node enables to position and orientate the child nodes which describe the visible geometry within the scene. Every node is detailed through fields. The orientation of the block Transform node is for example specified through its rotation field (Fig. 4.18). The visible geometry described in the block Transform node is defined by its children field. The child node of the block Transform node is of type Shape. It contains an appear-

ance field referring to an Appearance node that specifies the visual attributes, for example material and texture, to be applied to the geometry. The geometry field contains an Extrusion node which is specified by fields such as creaseAngle, crossSection and spine. The second cylinder Transform node is similar to the block Transform node and its content is therefore not displayed.

4.3.2 Scene graph

The VRML scene graph forms a hierarchical tree structure of nodes. Nodes are specified either through fields or through further owned nodes. The VRML document shown in Fig. 4.18 is composed of nodes which display a high level of detail. The document therefore displays high granularity. A one-to-one mapping of the VRML nodes into UML enables to easily recognize the VRML-specific content in the UML model.

The VRML node types, such as Shape or Extrusion, can be instantiated and are as a consequence described in UML as classes. Similar to the SolidWorks-specific geometric entity types, the classes corresponding to the VRML node types are invariant. Instead of being defined redundantly several times in different UML models, they are described only once in the VRML profile. The VRML node type-specific classes defined in the VRML profile form a VRML metamodel, which can be displayed in a class diagram. A selection of type-specific node classes is represented in the class diagram of Fig. 4.19. The VRML node types are described as UML classes and their fields as UML properties. The class diagram for example shows the different VRML geometric node types, such as Box or Extrusion, which share a generalization relationship with the GeometricNode class.

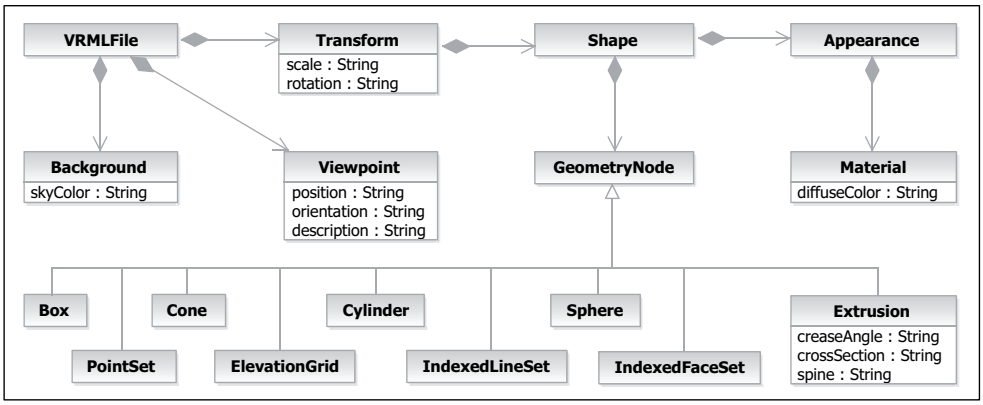

Figure 4.19: Selection of VRML type-specific node classes of the VRML profile

The VRMLFile class does not represent an existing VRML node. The VRMLFile class represents the container for all VRML nodes inside a VRML document. It can for example be composed of Transform nodes which themselves can be composed of Shape

nodes and so on. The possible composition of sample VRML nodes is displayed in the UML class diagram of the VRML metamodel (Fig. 4.19). It is common to all VRML files and as a consequence also to all VRML-specific UML models. Only a sample of the VRML node types is currently predefined in the profile. However, it is possible to dynamically create missing types in the UML model without needing to update the profile. These VRML-specific UML types need to be tagged with a *«vrml»* stereotype.

Every VRML node can be defined in UML through an instance. The composition of VRML nodes is therefore easily mappable into a composition of UML instances. Every UML instance in Fig. 4.18 for example belongs to at least one VRML-specific UML classifier predefined in the VRML profile. The block instance for example belongs to a class named Transform. Figure 4.18 exemplarily displays the composition of UML instances corresponding to the composition of nodes in the VRML file. The composition hierarchy composed of four levels is recognizable in the VRML file through the tabs and in UML through the linking of the instances (Fig. 4.18). In the example, the composition hierarchy from top to bottom includes the VRML file, the Transform node named block, the shape node and the extrusion node.

VRML nodes are described through fields such as spine in the case of the Extrusion node. VRML fields refer either to string values or to other nodes. Similar to VRML fields, UML instances can own values through UML slots which can refer to string values or to other instances. An instance slot thereby stores values according to the properties of the instance classifiers. The extrusion instance is for example defined through slots according to the creaseAngle, crossSection and spine properties of the Extrusion class. A UML slot refers to a string value through a UML literal string and to a UML instance through a UML instance value. In the slider example, the extrusion node refers to the creaseAngle through a literal string and the sliderMotion instance refers to the block node through an instance value (Fig. 4.18).

The sliderMotion instance has a special role as it represents the complete VRML model. Its classifier is therefore tagged with a *«vrmlFile»* stereotype. It owns properties named *url* and *header*. During an automatic translation from a UML model into a VRML file, the generated VRML file is saved according to the location specified in the url attribute. The optional header attribute defines the first line of the generated VRML file which is usually a comment detailing the VRML document type.

To avoid defining the same node several times, VRML nodes can be attributed a name acting as a node identifier. The required node can then be referenced through its name. This can for example be useful for an appearance node which needs to be reused. The node identifier is defined by the DEF keyword followed by the name. Contrary to UML

instances, not every VRML node has a name. The UML into VRML translation only exports UML instance names to VRML node names if the instances are tagged with a «$vrmlDEF$» stereotype. This is for example the case with the block instance (Fig. 4.18).

4.3.3 Assemblies

The VRML file content can be split into smaller separate VRML files. The smaller VRML files can then be reused in other VRML assembly files which only reference their embedded VRML files. The content of VRML assembly files is thus less detailed. The VRML assembly model of the slider mechanism can for example be composed of the separate base, crank, rod and slider VRML files, as shown in Fig. 4.20.

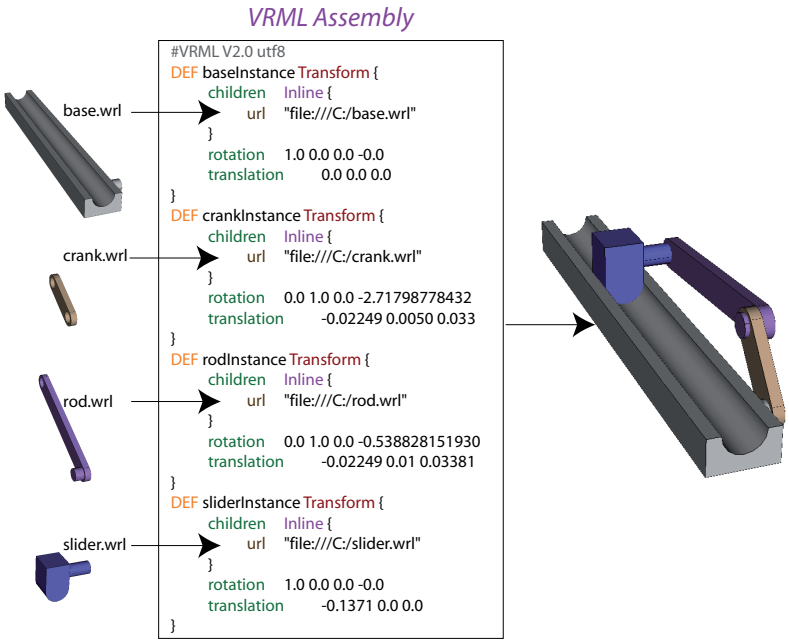

Figure 4.20: VRML assembly model composed of smaller VRML files

A VRML assembly file contains inline nodes which specify through Unified Resource Locator (URL) attributes the referenced embedded VRML files. An Inline node can be embedded inside a Transform node so that the referenced VRML content is placed within a VRML assembly file with a specific orientation and position. The corresponding values are described respectively through the rotation and translation fields of the Transform node. Consequently, the insertion of an embedded VRML file into a VRML assembly is fully defined by only three properties, namely the URL, the position and the orientation of the embedded VRML file. This allows to describe VRML assembly files composed of embedded VRML files in a corresponding UML model through VRML file-specific

instances instead of equivalent detailed VRML node-specific instances. As a result, the less detailed VRML assembly file can be mapped into UML with fewer instances.

The embedded VRML file is described in the VRML assembly file through a Transform node. It is thus described in UML, similarly to a CATIA or SolidWorks part, as a UML class with a *«vrmlTransformNode»* stereotype. UML constraints referring to property values can be more easily described and resolved than UML constraints referring to stereotype values. As the URL, position and orientation values of embedded VRML files often depend on other CAD models, they are described through properties of the *«vrmlTransformNode»* class and are tagged respectively with *«vrmlInlineURL»*, *«vrmlTranslation»* and *«vrmlRotation»* stereotypes.

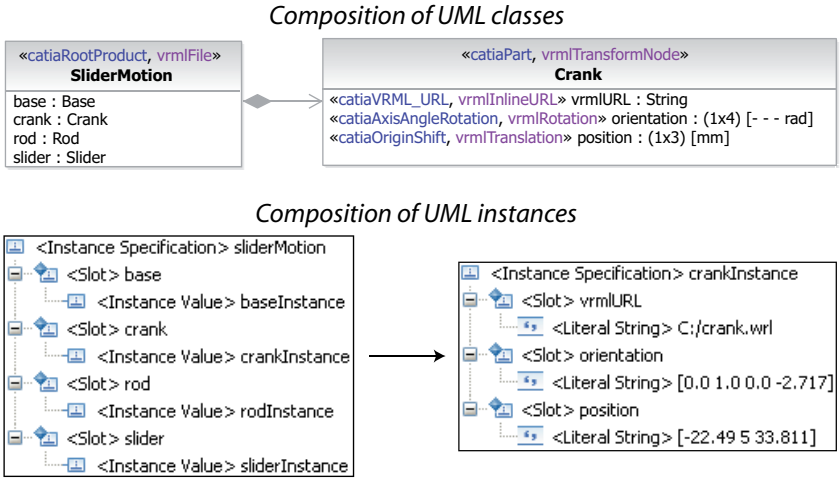

Figure 4.21: Selection of classes and instances corresponding to the VRML assembly

In the example of the slider-crank mechanism, the required stereotypes are displayed in the class diagram of Fig. 4.21 which only shows the SliderMotion and Crank classes. The SliderMotion class represents the VRML assembly file and is correspondingly tagged with a *«vrmlFile»* stereotype. The stereotype has *URL* and *header* properties for an automatic translation of a UML model into a VRML assembly file. The Crank class represents an embedded VRML file and is thus tagged with a *«vrmlTransformNode»* stereotype. The URL, the orientation and position of the crank-specific embedded VRML file are defined through UML properties which are respectively tagged with *«vrmlInlineURL»*, *«vrmlRotation»* and *«vrmlTranslation»* stereotypes.

Figure 4.21 exemplarily shows the slider mechanism instance named sliderMotion which owns the crank instance. The crank-specific content is for example loaded from a file situated at “C:/crankwrl” and positioned within the slider mechanism assembly at the position [-22.495 5 33.811] with an orientation described as quaternion equal to [-0.0 1.0 0.0 -2.717].

The description of VRML content in UML through predefined classes, as in the previous Section, or through a few stereotypes, as in this Section, is possible simultaneously. A hybrid approach is appropriate if a VRML file is composed of embedded VRML files as well as detailed VRML nodes. A UML model can then at the same time be composed of instances of stereotyped classes as well as of predefined VRML-specific classifiers.

4.3.4 VRML assembly files based on CATIA

A 3D geometry authoring tool offers professional editing functions, such as features and boolean operations, which are non existent in VRML. It is advantageous to use a 3D geometry authoring tool to edit a desired geometry and then to export it into the VRML format. Most CAD software applications support an export of geometric parts into the VRML format. In the example of the slider mechanism, the single base, crank, rod and slider VRML parts would ideally be defined in a 3D geometry authoring tool such as CATIA and then be exported into VRML.

In contrast to the export of single geometric parts, the export of complete CAD assembly models into VRML is not always possible. A VRML assembly file which is intended to correspond to a CAD assembly model can reference the exported single VRML parts and place them according to the position and orientation values of the parts within the CAD assembly model. However, a change in the geometry would require another sequence of time-consuming manual export, measure and copy & paste procedures.

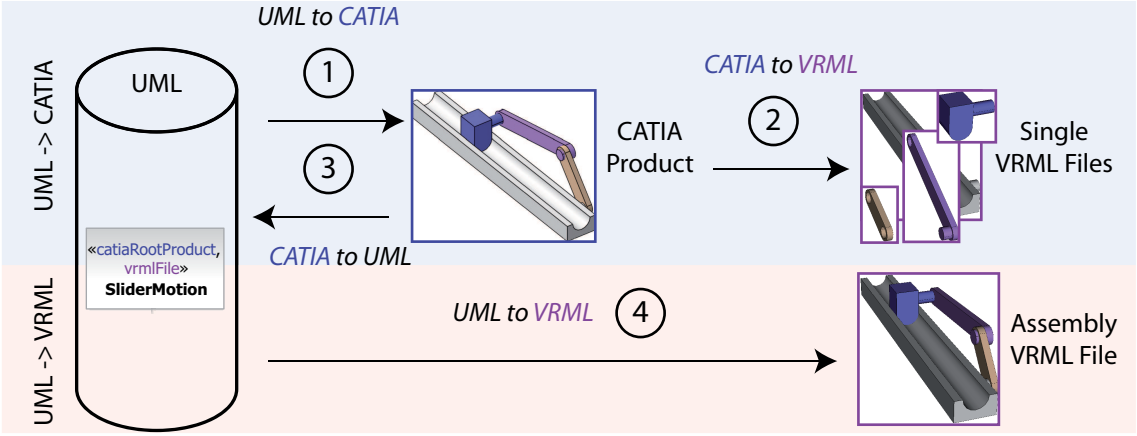

Figure 4.22: CATIA-based generation of a VRML assembly file via UML

The central UML-based product model can enable a quick automatic update of VRML assembly models based on CATIA assembly models. The CATIA and VRML representations of the slider mechanism parts are semantically equivalent in an abstract UML representation. Each mechanism part is therefore described in the UML model through

a class which is common to both the CATIA- and the VRML-specific geometry. The previously presented VRML- and CATIA-specific stereotypes can be applied in superposition onto the same UML classes and properties. Since the same UML element represents both CATIA- and VRML-specific geometry, data consistency between the different geometric formats is guaranteed. Both CATIA parts and VRML Transform nodes are described in UML through classes. Similarly, CATIA- and VRML-specific orientation and position values are described through UML properties. The UML classes and properties which refer to the same slider mechanism parts can therefore be tagged with CATIA- as well as with VRML-specific stereotypes as in Fig. 4.21. The SliderMotion class representing the complete assembly is for example tagged with a «$catiaRootProduct$» and a «$vrmlFile$» stereotype. The Crank class and all its properties are tagged with both CATIA- and VRML-specific stereotypes.

The process of automatically generating a VRML assembly file based on a CATIA assembly model via a common UML-based product model is described in Fig. 4.22. The first step consists of exporting the CATIA-specific geometry described in UML into a CATIA assembly model. Within the UML into CATIA translation process, CATIA then exports the newly generated CATIA parts into VRML files, as depicted in the second step. Step three returns the position and orientation measures of the newly generated CATIA part instances inside the CATIA assembly model back to the UML model. The UML model contains, after its interaction with CATIA, the required information to generate a VRML assembly file based on the single CATIA- generated VRML files and the CATIA-specific part orientation and position measures, as displayed in step four of Fig. 4.22.

4.4 Summary

Chapter 4 described the UML extensions required to represent in a UML-based product model the commonly shared geometric product information, such as volume, mass, center of gravity and moment of inertia, as well as geometric application-specific modeling concepts, such as parts, assemblies, assembly constraints, part dependencies, features and geometric primitives, in order to enable an automatic translation of UML-based geometric information into application-specific geometric models. The approach was investigated with modern and widespread 3D geometry modelers such as CATIA, SolidWorks and VRML.

The semantics of geometric modeling concepts conformed to the semantics of generalized object-oriented UML modeling concepts. Part and assembly models correspond to geometric templates which can be instantiated and inserted into other geometric models.

Features are recurrent geometric operations which equally represent templates that can be instantiated for the editing of detailed geometry. Geometric templates and template instances were therefore described in UML respectively through classes and class instances. As a result, part and assembly model attributes such as mass or volume were represented in UML through class properties, part interfaces through UML class interfaces, and assembly constraints through UML class constraints. The use of stereotypes on the generic UML modeling elements then sufficed to describe their application-specific geometric denotation.

However, the combination of a UML modeling element with an applied stereotype is not meaningful to describe a low-level geometric entity which is defined according to further geometric entities. A sketch element can for example be defined as being on a specific plane, which itself is positioned according to specific lines. The properties of a UML stereotype are not suited to describe the varied and detailed decomposition of a low-level geometric entity. Instead, a data model of the various geometric entity types was described in UML through a UML class for each geometric entity type such as Point, Line and Sketch. As the geometric data types do not change, they were predefined in their respective application-specific UML profile. Instances of geometric entity types, such as specific points, lines and sketches, were then represented accordingly as UML instances.

In summary, high-level object-oriented geometric modeling concepts were described in UML through corresponding generic UML modeling concepts with their respective stereotype and detailed low-level geometric entities were represented as instances of predefined geometric UML types. In both cases, the mapping between geometry-specific and generic UML modeling concepts occurred according to an easily understandable one-to-one correspondence.

Chapter 5

UML profiles for dynamic system models

Many real world problems in product design involve time-dependent processes. Their description often results in an accumulation of differential equations. A dynamic system is composed of mathematical equations to compute the time-dependent variance of system states. An example of a safety-critical dynamic system is an aircraft autopilot which automatically stabilizes an aircraft by adjusting aircraft control surfaces in case of disturbances. The algorithm of a controller is often adapted according to several criteria including stability, minimal energy consumption and quick reactivity. Algorithms are often first tested through the simulation of their corresponding dynamic system model and later implemented in code to be runnable on an embedded system. Section 5.1 presents the mapping of a Simulink-specific dynamic system model into UML.

A special kind of dynamic system is the multibody system composed of bodies which by mutual interaction follow translational or rotational displacements. The possible relative movement of each body to another is described by physical connections whose number and type influence the motion of the entire multibody system. An example of a multibody system is an industrial robot whose capacity to follow new movements for new tasks depends on the number of its arms and types of joints. Besides flexibility, its speed and energy consumption depend on its selected movement and its inertial properties. The simulation of a multibody system can for example be useful to compute the forces needed to set it in a desired motion, which in turn will determine the required power supply for the system. Following the standard Newtonian dynamics, a set of differential equations and constraint conditions can mathematically describe the motion of a multibody system. Section 5.2 describes the mapping of a SimMechanics-specific multibody system model into UML.

5.1 UML profile for Simulink-specific dynamic systems

Simulink[1] is used in many disciplines, from aerospace engineering to systems biology, to graphically model and simulate dynamic systems. It was therefore chosen in this research work as dynamic system modeling application to investigate the mapping of dynamic system modeling concepts into UML.

5.1.1 Simulink model

Simulink models represent relationships between system states in a block diagram. A dynamic system is described in Simulink by a block diagram consisting of blocks connected by lines. A wide range of blocks for signal sources, signal sinks, linear and non-linear components is available and the definition of custom blocks is also possible. The lines between blocks represent the flow of signal values. A simple dynamic system for example is represented in Fig. 5.1 top. The source block outputs a sinusoidal signal which is separated into two signals, one of which is integrated over time by the operation block. Both signals are regrouped as a single signal before ending in the sink block which saves and displays the incoming values over time during simulation. The incoming sinusoidal signal is displayed in blue and its integrated version in purple (Fig. 5.1 top).

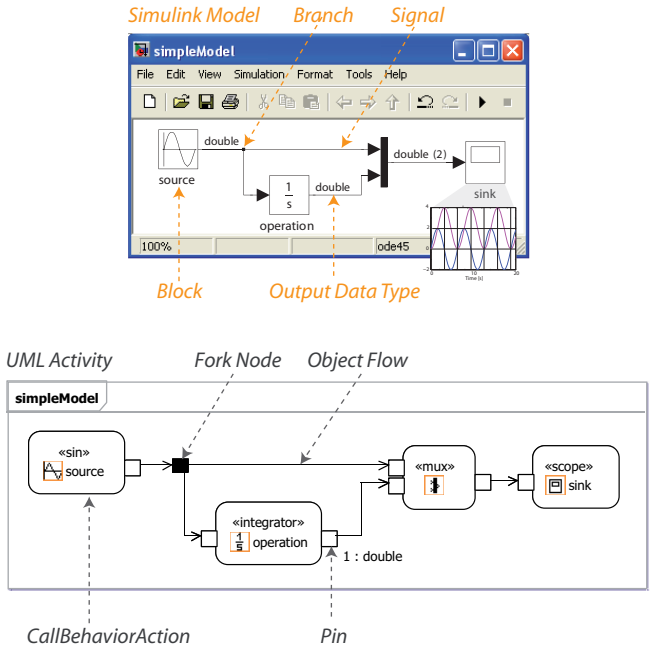

Figure 5.1: Simulink model and corresponding UML activity diagram

[1]The MathWorks, Simulink®, `http://www.mathworks.com/products/simulink/`

A Simulink model is composed of data flows between ports of Simulink blocks. These flows could be defined by UML connectors between ports of parts owned by a UML composite class describing the Simulink model. But the continuous streaming nature of Simulink data flows is better represented by UML object flows than UML connectors. This is underlined by the fact that the UML specification [122] proposes a *stream* text annotation to be applied on pins of object flows. Furthermore, the graphical similarity of Simulink signals is higher with UML object flows than with connectors. Simulink models containing blocks and signals are therefore mapped into UML activities consisting of actions and object flows. An example of a Simulink model and its corresponding UML activity is displayed in Fig. 5.1. The UML activity describing a Simulink model is tagged with a «$simulinkModel$» stereotype.

5.1.2 Blocks

Due to the frequent use of similar blocks responsible for similar computations, Simulink offers a library of standard block types which can be extended by user defined block types. Blocks inside a Simulink model are, strictly speaking, instances of predefined block types. The operation and sink blocks of the Simulink model (Fig. 5.1 top) for example are respectively instances of the block types Integrator and Scope. Each Simulink block represents a computation depending on its type and its tunable parameters. The source block instance for example outputs a sinusoidal signal according to its frequency and amplitude parameter values.

The relationship between block types and block instances is identical to the relationship between UML classes and UML instance specifications. As a Simulink block type represents a behavior, Simulink block types are mapped into UML activities instead of casual classes and Simulink block instances are interpreted as instance specifications having as classifier the corresponding activity. The most common standard block types of the Simulink library are translated as predefined activities of the UML Simulink profile. Simulink block type parameters are depicted as properties of their equivalent UML activities. Some parameters are common to all block types such as the position of a block inside a Simulink model. Common parameters are translated into properties of a higher level activity which is inherited by all other block type-specific activities of the Simulink profile.

The UML callBehaviorActions graphically represent the Simulink block instances and are tagged with a Simulink block type-specific stereotype. The Simulink block instance operation of the Simulink model in Fig. 5.1 for example is translated into a UML action tagged with an «$integrator$» stereotype standing for the Integrator block type.

The stereotype icon facilitates the recognition of the Simulink block type-specific actions in the UML activity diagram. As the Simulink block instance details are described in a related UML instance specification, each UML action references the related UML instance specification through an *activityInstance* property of the applied block type-specific stereotype. As the *activityInstance* property is common to all Simulink block type-specific stereotypes, the property is declared with a general stereotype from which all block type-specific stereotypes inherit.

The Simulink library offers a multitude of different block types, so not each one is mirrored as a predefined activity in the Simulink profile. In case a predefined activity to denote a Simulink block type is missing, a corresponding activity can be defined in the UML model. It is then tagged with a neutral «$simulinkBlockType$» stereotype while its owned block-specific properties are tagged with a «$simulinkBlockProperty$» stereotype. The related action is tagged with a «$simulinkBlock$» stereotype.

5.1.3 Signals

Simulink signals forward streams of values between Simulink blocks. The Simulink source block of Fig. 5.1 for example outputs values of type Double. UML object flows are edges between nodes that can have objects passing along them. Due to their similarity, Simulink signals are translated into UML object flows with an applied «$simulinkSignal$» stereotype. The Simulink signal label, if existent, is set equivalent to the name of the UML object flow. The Simulink input or output signals are attached to the ports of the Simulink blocks. The ports of the Simulink blocks are converted into UML pins as they specify the inputs and outputs of the UML actions. The «$simulinkSignal$» object flows then connect the pins. The source and target attributes of the object flows are therefore set to their corresponding connected pins and the incoming and outgoing attributes of pins are set respectively to their incoming and outgoing object flows. Each Simulink signal transmits one value per simulation time step. The *weight* attribute of the UML object flows, which determines the number of tokens consumed from the source node on each traversal, is therefore always set to 1. The *guard* attribute of the object flows, which determines if the object flow can be traversed, is always set to true.

The Simulink signal data type is equal to the block output data type which is double by default. Simulink blocks can output one-, two-, or multidimensional signals. The easiest is a scalar signal which consists of a stream of scalar values at a frequency of one scalar value per simulation time step. This is the case with the source and integrator blocks of Fig. 5.1. But signals can also consist of a stream of multidimensional vectors. The mux block in the same Simulink model for example receives two streams of scalar values

and combines them into a single vector output stream of dimension (1x2). The Simulink signal data type is translated as UML type of the source and target pins of the respective «$simulinkSignal$» object flow.

Simulink signals can have branches which split a signal into multiple signals. The output signal of the Simulink source block of Fig. 5.1 is for example separated into two signals directed at the operation and mux blocks. As a UML fork node splits a flow into multiple concurrent flows, a Simulink branch is depicted in UML as a fork node with a «$simulinkBranch$» stereotype. The incoming and outgoing attributes of the fork node are set to the incoming and outgoing «$simulinkSignal$» object flows. The Simulink mux block of Fig. 5.1 could be interpreted as a UML *join node*. The Simulink mux block combines its multiple inputs into one composite output signal. Simulink composite signals have no functional effect but can simplify the appearance of a Simulink model when many parallel signals exist. Similarly, a UML join node synchronizes multiple input flows into one output flow. In order to keep the resemblance between the UML and the Simulink model as high as possible, the Simulink mux block is translated like the other blocks into a UML action.

5.1.4 Subsystems

Large Simulink models can be decomposed into smaller models called subsystems which improve the overall overview of the modeled system. A subsystem is an encapsulated model which can be reused in the context of another Simulink model. A Simulink model can refer to an embedded subsystem via a block of type Subsytem. This can lead to a hierarchy of embedded models of any depth. The Simulink model of Fig. 5.1 can for example be decomposed as in Fig. 5.2 into a main and an embedded model. The main Simulink model contains a block of type Subsystem which refers to the embedded subsystem containing the previous integrator and mux blocks.

A Simulink subsystem is mapped like a Simulink model into a UML activity but tagged with a «$simulinkSystem$» stereotype. The «$simulinkModel$» stereotype is in fact a specialization of the «$simulinkSystem$» stereotype, as it represents the uppermost Simulink system in the hierarchy of a Simulink model. As a consequence, the «$simulinkModel$» stereotype is defined in the Simulink profile as having a generalization relationship with the «$simulinkSystem$» stereotype.

The Simulink subSystem block instance is mapped like the other block instances into a UML callBehaviorAction with an applied block type-specific «$subsystem$» stereotype. The Subsystem block type is mapped like other block types of the Simulink library as a predefined activity with subsystem-specific properties in the Simulink profile. The UML

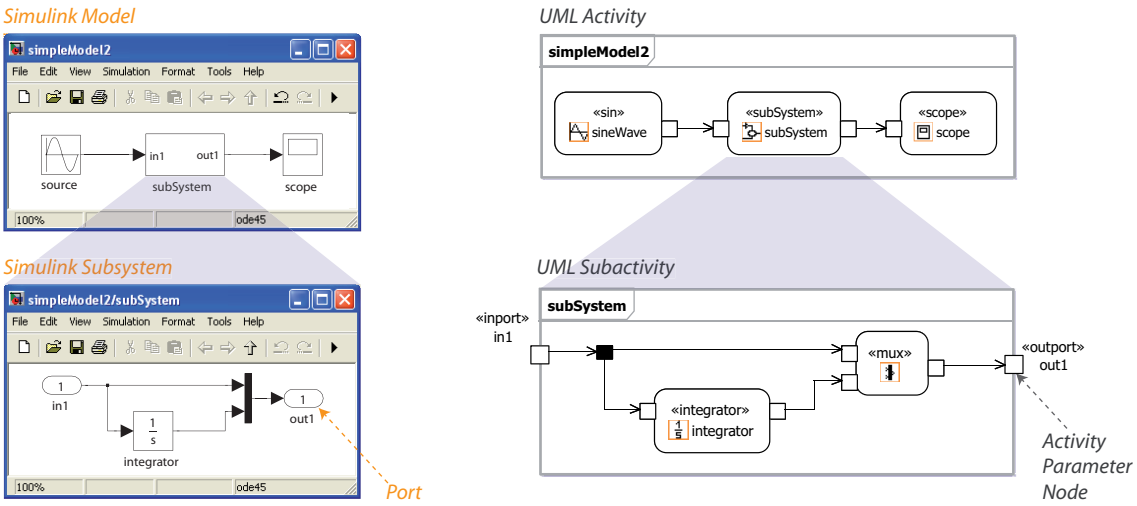

Figure 5.2: Simulink subsystem and related UML subactivity

action representing the Simulink subSystem block instance refers to an instance specification describing in detail exclusively the parameter values of the subSystem block instance while its *behavior* attribute refers to the embedded «$simulinkSystem$» activity.

Inputs and outputs of a Simulink subsystem are described by Simulink blocks of type Inport and Outport. Inputs and outputs of a UML activity are depicted by input and output activity parameter nodes. So the in- and outports of a Simulink subsystem are translated as UML activity parameter nodes of the *«simulinkSystem»* activity and are tagged accordingly either with an «$inport$» or an «$outport$» stereotype (Fig. 5.2 bottom right). The Simulink subsystem of Fig. 5.2 has for example an Inport block named “in1” and an Outport block named “out1”. The incoming or outgoing values of a Simulink subsystem are translated as UML parameters of the «$inport$» and «$outport$» UML activity parameter nodes. The *direction* attribute of UML parameters is set to In or Out according to their incoming or outgoing nature. The Simulink signal data type of the incoming or outgoing values of Simulink In- and Outport blocks is translated as the UML type of the corresponding UML activity parameter nodes and parameters.

5.1.5 Case study: slider position controller

An example of a dynamic system model applied to the slider-crank mechanism of Fig. 4.3 is a slider position controller. By applying a torque on the crank, the slider can be displaced along its track (Fig. 5.3). The controller is responsible for computing the torque to be applied on the crank so that the slider follows a target position. The Simulink dynamic system to simulate the slider position controller and the simulation results are shown in Fig. 5.4 left. The simulation results are displayed through the result Scope block (Fig. 5.4

top left) showing the target slider position as a green stepwise dashed line and the actual slider position as a purple continuous line. The simulation results show that the slider follows the target position with a slight overshooting at each position adjustment.

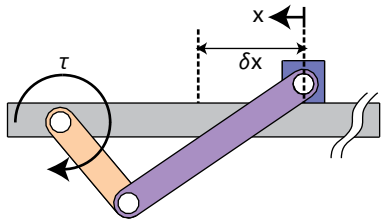

Figure 5.3: Displacement of the slider according to a torque applied on the crank

The target slider position is determined by a Simulink block of type RepeatedSequenceStair which switches the target position from 0 to 0.05 meters at every sample time of 1 second. The difference δx between the actual and the target slider position is forwarded to blocks of type Gain, Integrator and Derivative and summed up to determine the torque τ to be applied on the crank to minimize the slider positioning error δx. This most common type of controller is known as a proportional-integral-derivative (PID) controller. A saturator block limits the highest possible torque to be applied to 0.02 Nm to respect the limits of a simulated motor. The sliderMechanism block computes the new slider position as a consequence of the torque applied on the crank. It is described in this example as a black box but its internal structure representing a multibody system is described in the next chapter. The resulting actual slider position is forwarded back to complete the control loop and is also sent to the result block for visualization. The parameters of the Gain, Integrator and Derivative blocks can further be adjusted according to the requirements concerning controller stability, speed and energy consumption.

The UML activity corresponding to the Simulink model of the slider position controller is displayed in Fig. 5.4 right. As the Simulink Sum blocks show the operators acting on their signal inputs, the pins of the UML actions representing the Simulink blocks of type Sum have an attached UML keyword to display the corresponding sign.

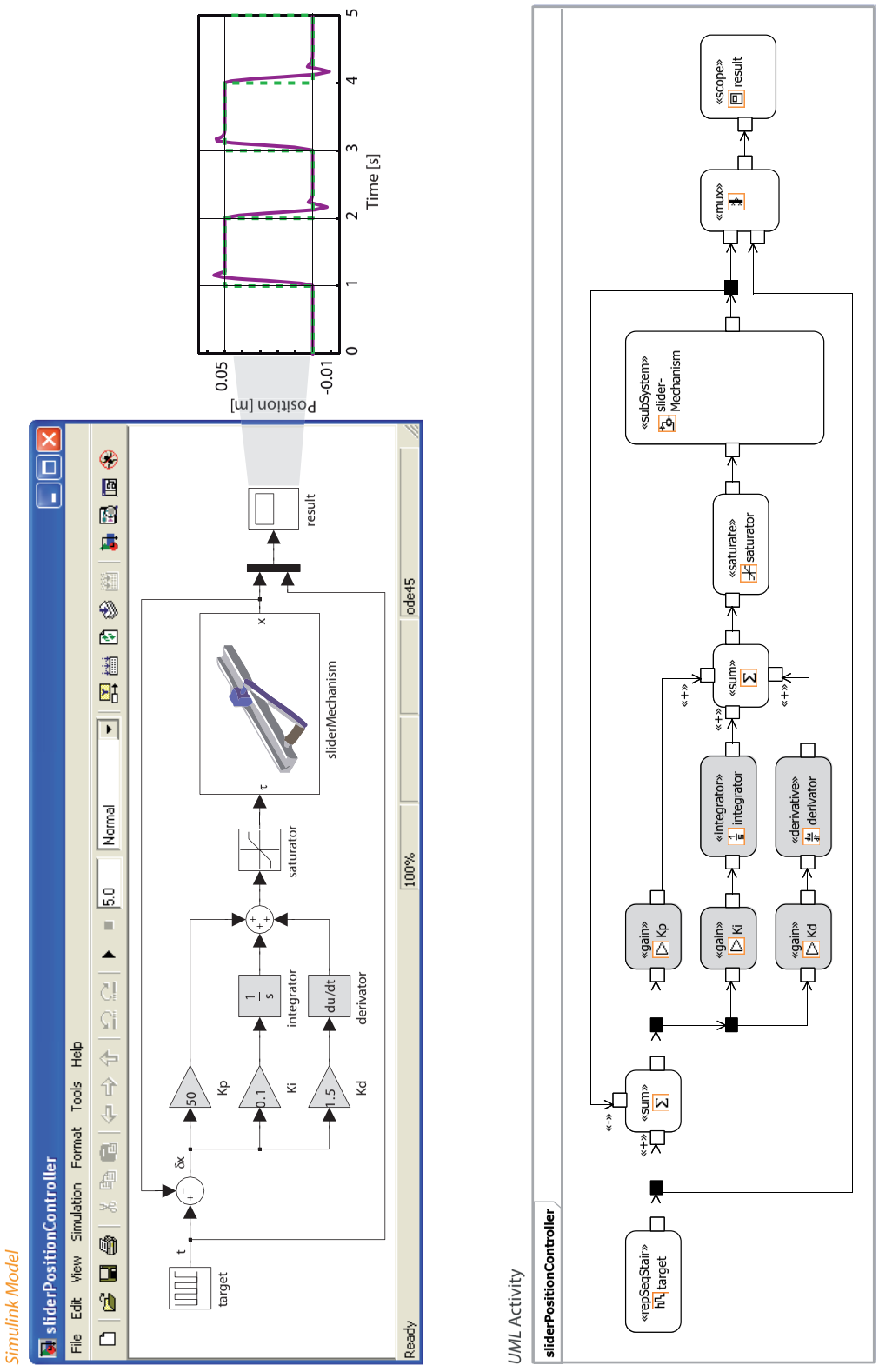

Figure 5.4: Simulink model of the slider position controller and corresponding UML activity diagram

5.2 UML profile for SimMechanics-specific multibody systems

SimMechanics[2] is an extension of the Simulink software and is specialized in the motion simulation of multibody systems. SimMechanics allows to represent a multibody system in a graphical model from which the mathematical equations to describe the multibody system motion are automatically derived and solved. SimMechanics thus greatly facilitates the error-prone process of establishing and solving equations of motion, especially in the case of mechanical systems composed of many bodies and joints.

5.2.1 SimMechanics model

A simple multibody system is for example the double pendulum of Fig. 5.5 formed of two cylinders connected by a revolute joint. The dashed lines show the trajectory of both cylinder ends when the cylinders are left to oscillate through the force of gravity for 0.5 seconds after being initially placed in the left starting position without velocity. The corresponding SimMechanics model is shown at the top of Fig. 5.7. A multibody system is described in SimMechanics as a block diagram similar to a dynamic system in Simulink. SimMechanics blocks represent in particular bodies, joints, force elements, sensors and actuators.

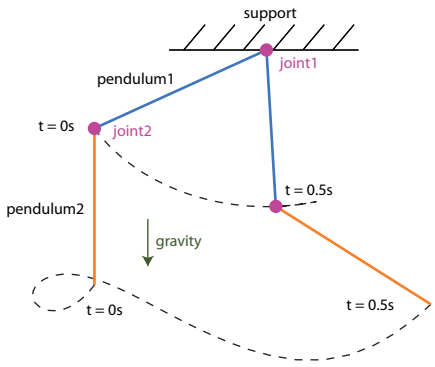

Figure 5.5: Trajectory of a double pendulum

A body is represented in SimMechanics by a block of type Body. Both pendulum bodies of Fig. 5.5 are for example described in SimMechanics respectively through the pendulum1 and pendulum2 blocks of type Body shown in Fig. 5.7. A body is specified in SimMechanics by its mass, its moment of inertia tensor and by coordinate systems which

[2]The MathWorks, SimMechanics™, `http://www.mathworks.com/products/simmechanics/`

are fixed to the body and move with it (Fig. 5.8). These attached coordinate systems are used to define the initial position of the bodies as well as the kinematical constraints between them. The body block corresponding to the pendulum1 of Fig. 5.5 for example needs three coordinate systems to be completely defined (Fig. 5.6). The CG coordinate system describes its initial center of gravity position and orientation and the CS1 and CS2 coordinate systems are needed for the definition of revolute joint constraints with their neighboring blocks, namely the support and the second pendulum block.

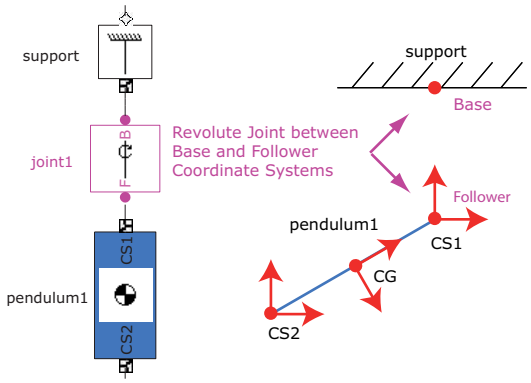

Figure 5.6: Coordinate systems for the representation of a double pendulum in SimMechanics

The kinematical constraints which restrict the motion of the bodies are specified in SimMechanics as blocks. For example, the revolute joints of Fig. 5.5 which restrict the motion of the pendulums are specified in SimMechanics as blocks of type Revolute Joint (Fig. 5.7). A joint block is connected with the constrained body coordinate systems. The SimMechanics joint2 block for example (Fig. 5.7) is connected with the coordinate system CS2 of the pendulum1 block and with the coordinate system CS1 of the pendulum2 block to define a revolute joint between the respective coordinate systems of both pendulum bodies. Furthermore, a SimMechanics model is always composed of a ground block to define a gravity force acting on the multibody system and an environment block to specify a fixed coordinate system (Fig. 5.7).

A SimMechanics model describes a multibody system by specifying its internal structure composed of bodies and connections. The SimMechanics modeling elements are therefore close to the concepts of internal structures in UML. A SimMechanics model is hence mapped into a UML class describing its internal structure through UML parts, ports and connectors. The class describing a SimMechanics model is tagged with a $\ll simMechModel\gg$ stereotype. An example of the UML composite structure corresponding to the SimMechanics model depicting the double pendulum is shown at the bottom of Fig. 5.7 .

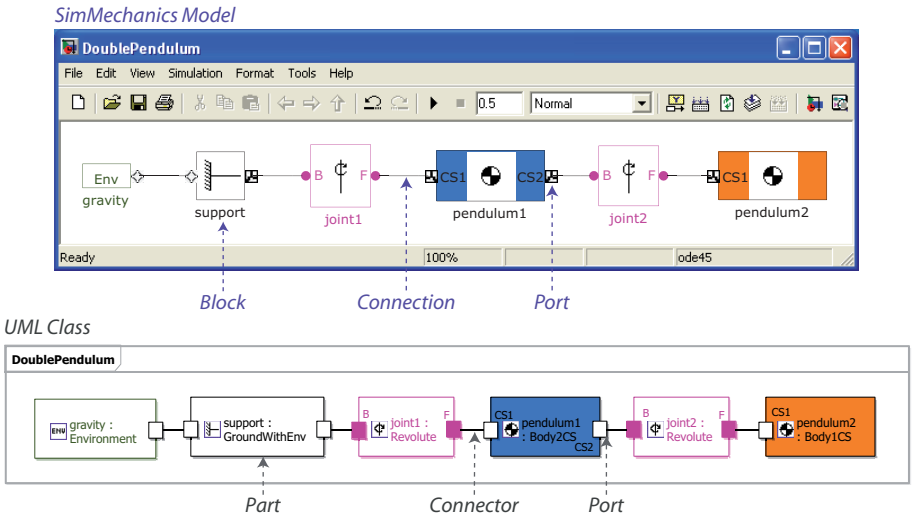

Figure 5.7: SimMechanics model of a double pendulum and corresponding UML composite structure diagram

5.2.2 Blocks

Similar to Simulink, SimMechanics has a predefined library of block types. SimMechanics blocks inside a SimMechanics model are instances of predefined SimMechanics blocktypes. The pendulum1 block (Fig. 5.7) for example is an instance of the blocktype Body. The relationship between a SimMechanics block type and a block instance in a SimMechanics model is identical to the relationship between a UML class and a UML instance specification. Therefore, a SimMechanics block type is translated into a UML class and a block instance into a UML instance specification. The frequently used block types of the SimMechanics block type library are mapped into predefined UML classes in the SimMechanics profile.

The containment relationship between a SimMechanics model and a SimMechanics block instance is described in UML on the class and on the instance level. On the class level, it is represented as a composite aggregation relationship between the UML class related to the SimMechanics model and the UML class related to the SimMechanics block type. The resulting property of the «$simMechModel$» class is called a part and is tagged for easier recognition with a SimMechanics block type-specific stereotype. The SimMechanics support block of type Ground is for example mapped into UML as a part tagged with a «$ground$» stereotype (Fig. 5.7 bottom). The application of the SimMechanics block type-specific stereotype is only visible through the block type-specific icon applied to the part. The containment relationship between a SimMechanics model and a SimMechanics block instance also applies on the instance level. The instance of the «$simMechModel$» class references the instance representing the SimMechanics block.

So in the example of the double pendulum of Fig. 5.7, the instance of the DoublePendulum «$simMechModel$» class references an instance of the Ground class playing the role of a support.

SimMechanics block instances inside a SimMechanics model are connected to each other via SimMechanics ports. Similarly, UML parts inside a UML class interact with each other through UML ports. The SimMechanics ports of a block type are therefore mapped into UML ports of the related UML block type-specific class. But the ports of SimMechanics block instances are not always determined through their block type but sometimes also by the block instances themselves. The definition of UML ports on the instance level is however not possible in UML, as UML ports can only be defined as class attributes and are automatically valid for all instances without exception. To overcome this problem, the UML classes in the SimMechanics profile depicting the SimMechanics block types do not own ports and new UML classes are introduced in the UML model to represent specialized versions of the UML block type-specific classes with additional ports if needed.

The pendulum1 and pendulum2 blocks are for example both of the same block type Body but do not have the same number of ports, as pendulum2 has one port less. The UML parts corresponding to the pendulum blocks are respectively of type Body2CS and Body1CS. Both new classes inherit from the Body class of the SimMechanics profile but have a different number of ports. The class Body2CS owns three ports (Fig. 5.8 top right) as the pendulum1 part owns two ports representing user defined coordinate systems and one port representing the center of gravity coordinate system. The UML ports are either tagged with a port-specific stereotype such as «$simMechCS$», standing for SimMechanics coordinate system, or «$simMechCG$», standing for SimMechanics center of gravity, or with a general «$simMechPort$» stereotype. The type of the SimMechanics port is reflected as type of the UML port. The possible port types are defined in the SimMechanics profile.

A body coordinate system is defined in SimMechanics by its origin and orientation relative to another existing coordinate system. These SimMechanics attributes are displayed in the pendulum1 block parameters window in Fig. 5.8 left. The UML attributes describing such a coordinate system are also grouped together and belong to the UML class SimMechCS which is the type of the ports representing the body coordinate systems. The SimMechCS class is saved in the SimMechanics profile and is displayed in Fig. 5.8, listing for space reasons only the attributes concerning the initial coordinate system position. The type of some attributes is an enumeration also belonging to the SimMechanics profile. The componentsInAxesOf property is for example of type CSEnum. This is an

enumeration consisting of literals such as "*Adjoining*" which specifies an equivalence with a neighboring coordinate system.

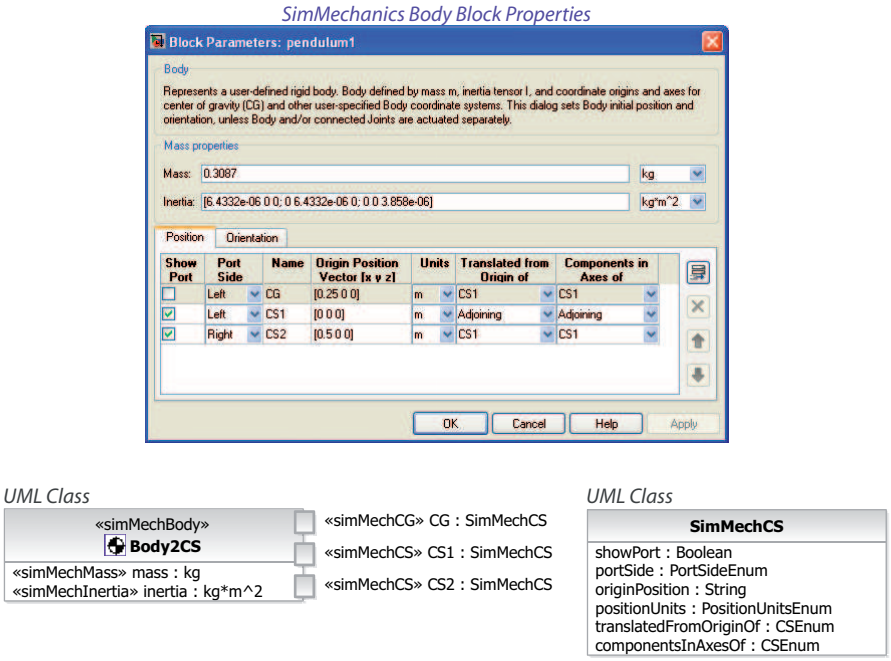

Figure 5.8: Top: Properties of the pendulum1 Body Block. Bottom: UML Classes corresponding respectively to the Body blocktype and the coordinate system port type

Important SimMechanics block properties such as the mass and inertia of a body block are directly editable in their block parameter windows (Fig. 5.8). The distinction between the UML properties corresponding to the essential block attributes and the others which are not visible in a SimMechanics block parameters window is helpful. This distinction is possible by introducing into the UML model new UML classes which inherit from the UML SimMechanics blocktype-specific classes of the SimMechanics profile. This specialization is in fact already necessary due to the ports. The new classes redefine, if considered helpful, the important properties so that these and only these are then visible in the UML model. The redefined properties are tagged with an attribute-specific stereotype such as the mass property of the Body2CS class which is tagged with a «$simMechMass$» stereotype (Fig. 5.8 top right). The multidisciplinary role of some redefined properties can then subsequently be captured by superpositioning other application-specific stereotypes on these properties. This will be demonstrated in the slider-crank mechanism case study in Section 5.2.5.

5.2.3 Connections

Interactions between SimMechanics blocks inside a SimMechanics model are described through connections displayed as undirected lines between ports of blocks. Interactions between corresponding UML parts are specified through UML connectors between UML ports of UML parts. So the SimMechanics connections are mapped as UML assembly connectors tagged with a «$simMechConnection$» stereotype. The connectors between ports are valid according to the UML specification if both ports concerned have a common interface. One of the ports concerned must provide this interface and the other must require it. UML interfaces have therefore been defined in the SimMechanics profile.

The coordinate system ports belonging to the body blocks for example provide coordinate system data which is required by the coordinate system ports belonging to the joint blocks. A common CoordinateSystem UML interface has for this case been defined in the SimMechanics profile. The CoordinateSystem interface is provided by the body block SimMechCS ports and is required by the joint block ports (Fig. 5.6). The CoordinateSystem interface is the most common but similar interfaces, together with the application of interface realizations and usage dependencies, have also been defined for other cases.

If the UML connectors belong to a UML component instead of a UML class, the composite structure diagram can display through the “ball-and-socket” notation the requiring or providing role of parts such as in Fig. 4.8. But to keep the visual similarity between the SimMechanics model and the UML composite structure diagram as high as possible, the context classifier is a UML class showing the connectors without “ball-and-socket” notation.

5.2.4 SimMechanics model as a Simulink subsystem

Multibody systems are often set in motion through controlled actuators. It is therefore useful to combine a multibody system model with a controller model to test and simulate a complete system. A SimMechanics multibody system model can be combined with a Simulink controller model in one common Simulink model. SimMechanics blocks of type Actuator can receive Simulink signals describing a force or a torque and forward them to SimMechanics blocks describing a body or a joint. Reciprocally, SimMechanics blocks of type Sensor can measure the motion of a body or a joint and output it as a Simulink signal. Due to the decision of mapping Simulink blocks and signals into UML activity diagram elements and SimMechanics blocks and connections into UML composite structure diagram elements, a Simulink model containing on the same modeling level both Simulink and SimMechanics information cannot be mapped into one common UML

diagram. Elements of an activity diagram cannot contain parts and connections and a composite structure diagram cannot contain actions and object flows.

It is therefore necessary to separate the SimMechanics blocks from the Simulink blocks before converting them to UML by introducing a Simulink subsystem containing exclusively SimMechanics-related information. Only then can the combined Simulink/ SimMechanics model be translated into UML. As described in Section 5.1.4, a UML callBehaviorAction representing a Simulink subsystem block refers to a UML activity representing the related Simulink subsystem. The Simulink subsystem which contains a SimMechanics model is mapped into a UML subsystem activity which is set as *classifier behavior* of the UML class representing the SimMechanics model. The UML subsystem activity is empty and is used to link the UML action representing a Simulink subsystem with the composite structure of a UML class representing the SimMechanics model. In the example of the slider position controller of Fig. 5.4, the sliderMechanism block is a Simulink subsystem block referring to an embedded SimMechanics Model.

The SimMechanics in- and outport blocks are translated as UML delegation ports of the related *«simMechModel»* class and are tagged with corresponding *«inport»* or *«outport»* stereotypes. A UML interface named SimMechData has been created in the SimMechanics profile to describe the flow of data through the UML delegation ports. The types of the delegation in- and outports respectively require and provide the SimMechData interface. The directed signals originating or ending in the in- and outport blocks of the SimMechanics model are translated as directed UML delegation connectors.

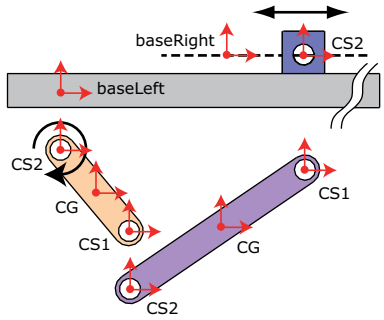

Figure 5.9: Schematic 2D view of coordinate systems for the abstraction of the slider-crank mechanism in SimMechanics

5.2.5 Case study: slider-crank mechanism as multibody system

The slider-crank mechanism of Fig. 4.3 is an example of a multibody system. The schematic 2D representation of the decomposition of the slider-crank mechanism into bodies and coordinate systems for the representation of the multibody system in Sim-

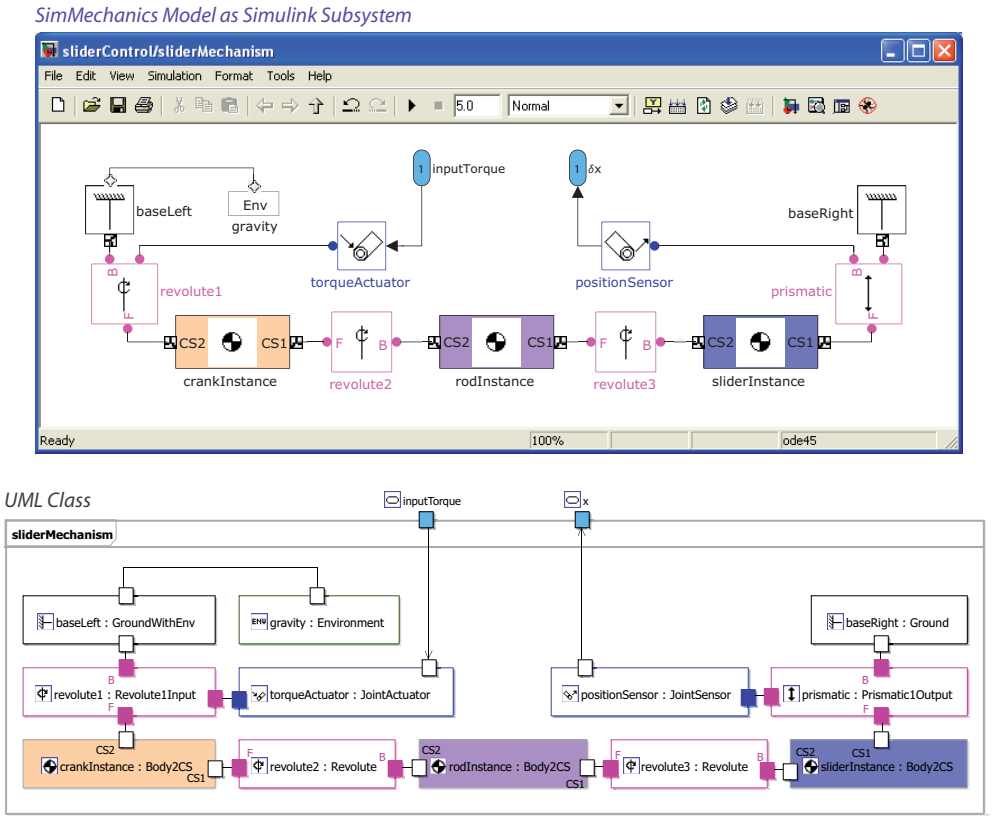

Figure 5.10: SimMechanics model of the slider-crank mechanism as Simulink subsystem and corresponding UML composite structure diagram

Mechanics is shown in Fig. 5.9. The corresponding SimMechanics model is represented in Fig. 5.10. The crank, rod and slider parts are represented in SimMechanics as body blocks. The base part is fixed and is represented by two ground blocks. Joint blocks define the possible movement of each body relative to another. Coordinate systems are introduced as in Fig. 5.9 to define the interaction points and positions of the bodies. The crank body is for example described by three coordinate systems. The CG coordinate system stands for the position of its center of gravity and the CS1 and CS2 coordinate systems represent the interaction points with the neighboring bodies such as the rod and the base. The kinematical constraint to limit the movement of the crank relative to the rod to a rotation around a common axis is defined by a constraint of type Revolute between the CS1 coordinate system of the crank and the CS2 coordinate system of the rod. This constraint is represented in the SimMechanics model by a revolute joint block named revolute2 connecting both constrained coordinate systems.

The SimMechanics multibody system model of the slider-crank mechanism is combined with a Simulink controller model to simulate a controlled behavior of the mechanism. The Simulink controller model presented in Section 5.1.5 adjusts the position of

the slider by rotating the crank. The slider-crank mechanism is described in Simulink by a subsystem block named sliderMechanism (Fig. 5.4) having as input signal a torque value and as output signal a position value. The underlying subsystem consists of the SimMechanics model of Fig. 5.10. The SimMechanics in- and outports are visible as UML delegation ports in the underlying SimMechanics model. The incoming signal is forwarded to a joint actuator block connected to the revolute1 joint which applies a torque on the CS2 coordinate system of the crank to make it rotate. Similarly, the position of the CS1 coordinate system of the slider is measured by a joint sensor block connected to a prismatic block. The corresponding UML class composite structure is displayed at the bottom of Fig. 5.10.

5.3 Summary

Chapter 5 has described the UML extensions required to represent in a UML-based product model Simulink-specific dynamic models as well as SimMechanics-specific multibody system models. Simulink-specific dynamic system models and SimMechanics-specific multibody system models are respectively similar to UML activity diagrams and UML composite structure diagrams. As shown in Fig. 5.4 and Fig. 5.10, the graphical resemblance between Simulink models and UML activity diagrams as well as between SimMechanics models and UML composite structure diagrams is high.

The Simulink-specific dynamic system model is a block diagram composed of blocks, signals and subsystems. As the Simulink dynamic system model as well as the subsystems represented specific behaviors, they were mapped into UML activities. The Simulink signals represented information flows and were accordingly mapped into UML object flows. Simulink block types represented templates and were described as predefined UML activities within the Simulink profile. Simulink block instances were mapped into both UML actions and UML instances to respectively graphically depict the block instances within UML activity diagrams and capture the values of the block instances. The UML actions thereby referenced the UML instances through a stereotype property. Ideally, block instances should be mapped into a single UML modeling element which would unite the characteristics of both UML actions and instances.

The SimMechanics-specific multibody system model is also a block diagram, however composed of blocks, ports and connections. The connections represent static dependencies instead of dynamic information flows such as in Simulink. The multibody system model was represented as a UML composite structure diagram. SimMechanics block types were thereby translated into UML predefined classes and connections into UML

connectors. As with Simulink block instances, the direct one-to-one mapping of a SimMechanics block instance into a corresponding UML modeling element is not possible. The block instances were mapped into UML parts and UML instances to respectively graphically depict block instances within the UML composite structure diagram and capture the values of block instances. The direct mapping of a SimMechanics block instance into a corresponding UML modeling element is not possible because UML composite structure diagrams describe the internal structure of a class of objects and do not directly refer to concrete instances.

In contrast to Simulink and SimMechanics models which describe interactions between block instances, UML object diagrams do not support the modeling of complex interactions between objects as they are intended to describe static snapshots of software runtime objects and their links. UML object diagrams with improved capabilities to describe interactions between objects would facilitate a one-to-one mapping of dynamic and multibody system models into UML. Future UML releases may therefore introduce new modeling concepts for a better representation of object interactions.

The mapping of Simulink, SimMechanics and combined Simulink/SimMechanics models into UML was applied to the slider-crank mechanism example. As the blocks within the Simulink and SimMechanics models contain detailed information, importing the existing models into UML was preferable to describing in UML the Simulink- and SimMechanics-specific information from scratch. The translation of UML models into Simulink and SimMechanics models was validated by generating models identical to the imported ones. Despite the inability of the UML to represent detailed interactions between instances, the dynamic and multibody system models shared great resemblance with their corresponding UML diagrams and therefore allowed a mostly intuitive one-to-one mapping.

Chapter 6

UML profiles for data retrieval and constraint processing

Products are often composed of parts which are ready-made and available off-the-shelf. The characteristics of these components, such as their dimensions, are often stored in spreadsheets. Section 6.1 presents the mapping of Excel-specific spreadsheet data into UML. A central product model also needs to reference data resulting from external program-specific computations. In this case, it must represent functions with their input and output arguments. Section 6.2 describes the mapping of Matlab-specific functions into UML. Furthermore, relationships between features of distinct models can be represented in a central product model through algebraic equations. Section 6.3 presents UML constraints to describe algebraic equations and their resolution.

6.1 UML profile for Excel-specific spreadsheet data

Excel[1] is a widely used spreadsheet application. It displays cells organized in rows and columns, each cell containing data or a formula with relative or absolute references to other cells. Excel furthermore has a multitude of graphing possibilities, which allows numerical data to be interpreted as graphs or charts. Product data is often stored in Excel documents which contain several spreadsheets. In the example of Fig. 6.1, the width and length values of crank parts are stored in an Excel spreadsheet. Each cell is identified through its location in the spreadsheet. The spreadsheet columns are numbered alphabetically while the rows are sorted numerically. The crank length which applies to all crank instances is for example saved in cell C2 while the crank instance-specific width values are respectively stored in cells B3 and B4.

[1] Microsoft Office Excel, `www.microsoft.com/excel/`

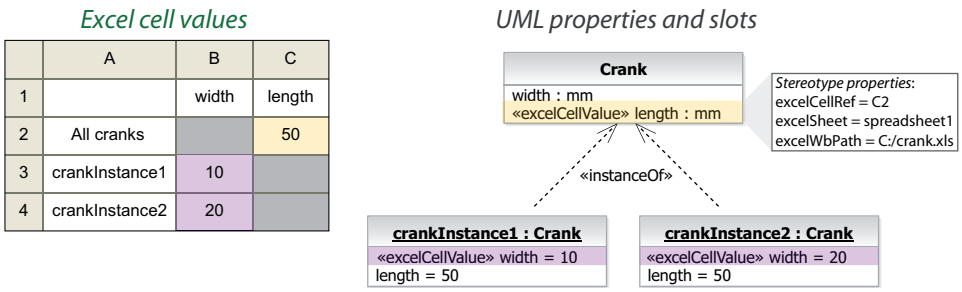

Figure 6.1: Excel cell values and corresponding UML properties and slots

A UML class represents the properties which are common to a set of instances. A value which is common to all instances is therefore set as default value of the corresponding property. The Crank class of Fig. 6.1 for example has a width and a length property. According to the Excel spreadsheet, the length value, colored yellow in the spreadsheet, applies to all crank instances. The default value of the length property of the Crank class therefore needs to be linked with the respective Excel cell. The property is then tagged with an «$excelCellValue$» stereotype which has three attributes to specify the cell location, the specific spreadsheet and the location of the Excel document which is also called workbook. Through the tagged property, the Excel length value has been transferred to each instance, as displayed in the length slots of the crank instances. On the other hand, the Excel width values which are colored purple only apply for specific instances. In UML the crank instance-specific values are stored in their respective UML slots. In this case, the same «$excelCellValue$» stereotype is applied to the slot to link the slot value with the Excel cell value. Both UML crank instances therefore have a width slot tagged with an «$excelCellValue$» stereotype which refers to an Excel cell value.

The presented Excel-specific stereotype is responsible for transferring the Excel cell values to the UML model. Other stereotypes can handle the reverse transfer of UML values to an Excel document. The method of applying a stereotype on properties or slots can also be used to link UML values with values stored in other data sources such as databases.

6.2 UML profile for Matlab®-specific functions

The MATLAB application offers an integrated environment supporting computation, visualization and programming. MATLAB allows to solve many technical computing problems, especially those with matrix and vector formulations, as it includes libraries which offer state-of-the-art software for matrix computation. The name MATLAB stands for matrix laboratory. The MATLAB programming language is a high-level language and in-

cludes matrix-based data structures, control flow statements, functions and object-oriented programming features. Collections of MATLAB functions (M-files) are grouped in toolboxes, as for signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation and many other domains.

The results of Matlab-specific computations can have an impact on other product data. By representing the Matlab-specific computations in a UML-based product model, the Matlab-specific data can be linked with other application-specific data. This Section shows the representation of a Matlab function in a UML model. An example of a Matlab function is displayed in Fig. 6.2. The header declares the function name getSliderMaxSpeed, its input argument simTime and output argument maxSpeed. The Matlab function will launch a simulation of the sliderControl Simulink model for 10 seconds. The Simulink model simulates the control of the slider-crank mechanism as in Section 5.1.5 and records the speed of the slider along the rail. Figure 6.2 shows the Simulink model which has a few additional blocks for the speed measurement. The maximum speed of the slider is recorded at specific time intervals in an array named maxSpeedRecord. The maximum speed which occurred during the simulation is equal to the last recorded value in the array and is set equal to maxSpeed which is the output argument of the function.

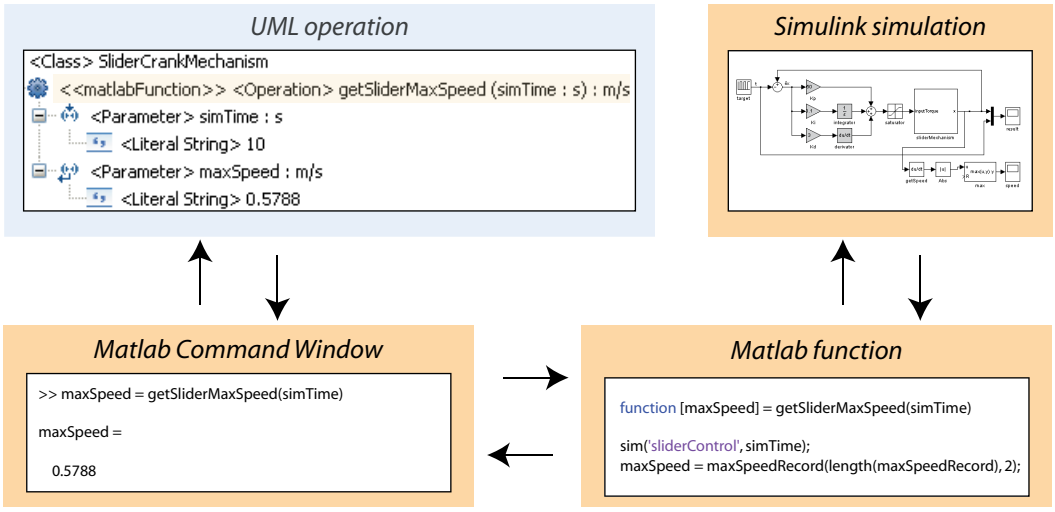

Figure 6.2: Processing of a UML operation referring to a Matlab function

Functions or methods are represented in UML through operations. The input and output arguments of an operation are specified through parameters. The getSliderMaxSpeed operation for example has a simTime input argument which is represented in UML through a parameter with a direction attribute set to “in”. The output argument maxSpeed is mapped into a parameter with a direction attribute set to “return”. The parameter values are described through literal strings. The input argument simTime is for example set to 10s

and the output argument maxSpeed, resulting from the computation by the Matlab function, is set to 0.5788m/s. The Matlab-specific function is recognizable in the UML model as an operation tagged with a *«matlabFunction»* stereotype. The name of the UML operation is then identical to the name of the function. The directory in which the Matlab function is saved is specified in the workspacePath attribute of the *«matlabFunction»* stereotype. It can also specify through an order attribute the sequence in which several Matlab functions need to be evaluated.

Figure 6.2 shows a tree view of the UML operation modeling elements including the input and return parameters and their values. According to the UML specification, UML operations can only have one parameter with the direction kind "return". This is the case in many programming languages such as Java or C++. However, Matlab functions can have several output arguments. In this case, the output arguments are mapped into UML parameters with the direction kind "out". UML operations are generic modeling elements which can represent functions of different programming languages.

6.3 UML profile for constraint processing

Dependencies between design variables can be described through mathematical statements to ensure data consistency. The mathematical statements can describe a multitude of different relationships between design variables such as inequations or differential equations. The most common type of mathematical statements during product design are equations which describe an equality between two expressions composed of design variables.

Equations can be described either in an explicit or an implicit form. Explicit equations describe the evaluation of a parameter based on a known function and a set of known variables in the form $x_n = f(x_1, x_2, ..., x_{n-1})$. If the explicit representation allows to compute one specific unknown variable, the engineer is forced to decide which variables are input and which are output. According to the example, only the x_n variable would be computed based on the other known variables. This is not practical in conceptual design as it requires the analysis of different what-if scenarios to explore the possible design space [136]. Trade-off studies can only be conducted by permutating the known and unknown variables.

The implicit or declarative representation is of the form $f(x_1, x_2, ..., x_n) = 0$ and does not visually impose a specific unknown variable. The implicit representation can be reused in different scenarios as the definitions of the known and unknown variables can be swapped. However, the explicit representation does not necessarily imply an explicit

resolution of the equation according to only one specific unknown variable. Many mathematical toolboxes can automatically transform one explicit representation into another in order to solve an equation according to another set of known and unknown variables. The sequence steps to solve an equation, or an equation system, are as a consequence not predetermined and do not stand for explicit computations. The resolution order of the equations is then not fixed but computed according to the known and unknown design variables. The simple equations presented in Fig. 6.3 are represented in an explicit form but are handled and manipulated by mathematical toolboxes as if they were in a declarative form.

Design variables are described as UML properties. UML constraints can set conditions or restrictions on UML properties. They are for example displayed in light red next to the classes in Fig. 6.3. The UML constraints describe equations which affect the length and thickness parameters of the crank and rod parts of the slider-crank mechanism. The UML constraints are composed of UML opaque expressions to describe the algebraic equations.

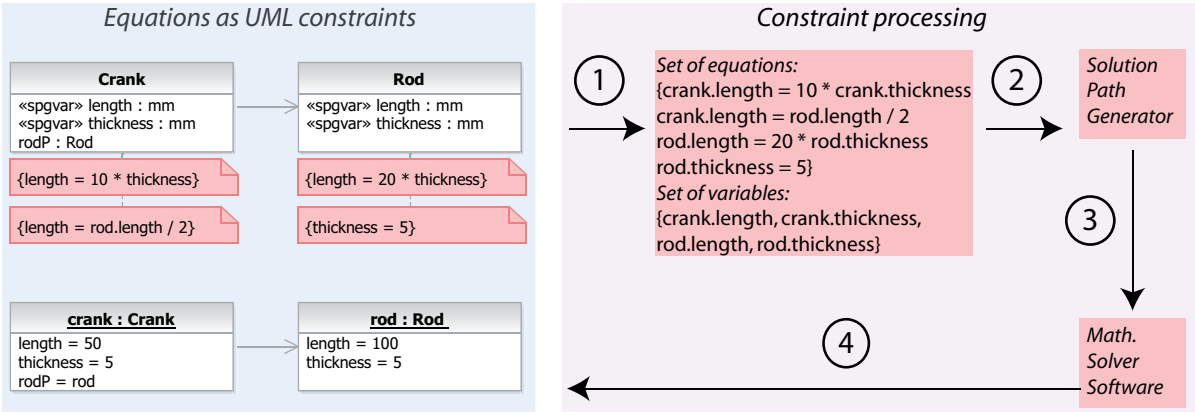

Figure 6.3: Resolution of equations described as UML constraints

By referring to the properties of a class, the constraints are in force for all instances of that class. So a constraint definition based on class properties can lead to a multitude of instance-specific constraints. These are then composed of unique instance-specific variables which are not to be mistaken for class-specific properties. The transformation of class-specific into instance-specific constraints is visible in the first step of Fig. 6.3 which displays the resolution process of the UML-based equations. The constrained variables have in addition to the property name a prefix depending on the instance name in order to create a unique variable. As an example, the Crank class-specific constraint $length = 10 * thickness$ is transformed into the crank instance-specific constraint $crank.length = 10 * crank.thickness$.

Constraints can refer to properties of a class and also to properties of referenced classes. As displayed in Fig. 6.3, the length of the crank depends on the length of the referenced rod. The constraint refers to the length property of the rod via the rodP property of the Crank class. The length property of the rod is transformed into a unique instance-specific variable when replacing the class-specific constraints with instance-specific constraints. For example, the class-specific constraint $length = rodP.length/2$ is replaced with the crank and rod instance-specific constraint $crank.length = rod.length/2$.

The automated resolution of UML-based equations consists of several steps as described in Fig. 6.3. The first step consists of generating instance-specific equations based on property-specific equations. The set of instance-specific equations can be directly sent to a mathematical solver. However, the solution path generator (SPG) [142], which is based on a bipartite matching algorithm acting on a graph representation of dependencies between equations and constrained variables, is able to compute the resolution order of the equations. The solution sequence of the equations is therefore determined in a second step before solving the equations by a computer algebra system such as Matlab[2], Mathematica[3] or Maple[4] in a third step. The computed results are sent back to the UML model in the slots of the corresponding instances, as displayed in step four in Fig. 6.3.

The unresolved properties which are involved in equations are tagged with a «$spgvar$» stereotype while the constant properties are tagged with a «$spgconst$» stereotype. The abbreviation "spg" stands for solution path generator and refers to the algorithm used to compute the equation solving sequence for a set of declarative equations. Although UML constraints can refer directly to the constrained properties through their constrainedElement attribute, tagged properties allow a better visualization of the equation variables.

Each UML opaque expression specifies the language of the expression. One predefined language for writing constraints is the Object Constraint Language (OCL) [114]. It is a query language predominantly used to specify values and to define pre- and postconditions regarding the execution of operations. However, the OCL does not support and interpret mathematical operands and is therefore not suitable to solve mathematical equations. As the resolution of algebraic equations is based on the common preprocessing SPG algorithm which is independent of the postprocessing mathematical solver, the equations are stored in UML expressions with the language "spg".

Other languages can be defined for the resolution of other types of equations. An example of an equation consisting of matrices is described in Fig. 6.4. Position or inertia values are often represented through matrices. The transformation of a position vector

[2]The MathWorks, Matlab, `http://www.mathworks.com/products/matlab/`
[3]Wolfram Research, Mathematica, `http://www.wolfram.com/products/mathematica/`
[4]Maplesoft, Maple, `http://www.maplesoft.com/products/Maple/`

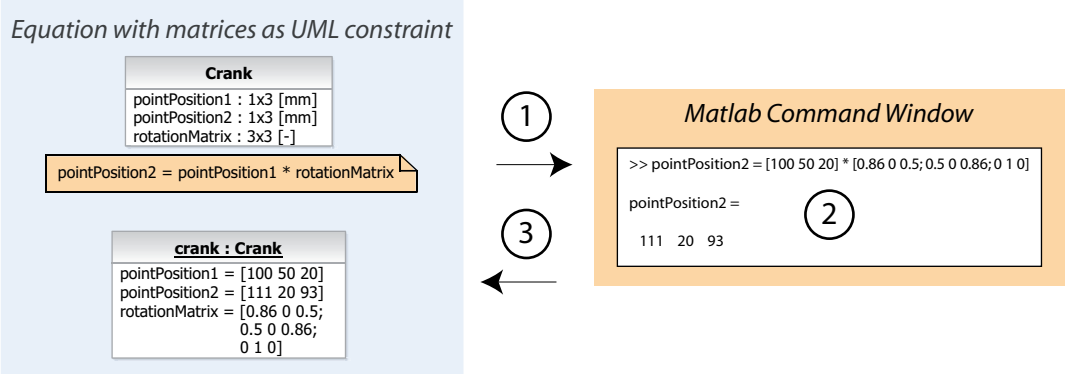

Figure 6.4: Resolution of equations involving matrices described as UML constraints

into another reference frame can occur by multiplying it with a transformation matrix. The equation in Fig. 6.4 describes the relation between the position vectors pointPosition1 and pointPosition2 respectively in the reference frames indexed 1 and 2. The equation is described in UML by an opaque expression owned by a constraint. In this case, the equation is represented in an explicit form and the resolution of the equation is explicit. The left side of the equation represents the unknown variable pointPosition2 while the right side contains the known variables pointPosition1 and rotationMatrix. The known variables are replaced with their values and the equation is directly sent to a mathematical solver such as Matlab (step 1). The result of the Matlab computation (step 2) is sent back to the UML model and placed in the corresponding UML slot (step 3). As the UML expression describing the equation is directly solved by Matlab, its language is correspondingly set to Matlab. In the case of several Matlab-specific equations, a supplementary stereotype can be applied on the UML expression to specify the order in which the explicit equations are to be solved.

6.4 Summary

Chapter 6 has shown the retrieval of data from data sources such as Excel spreadsheets. The UML-based reference to spreadsheet data occurs with stereotypes that are applied either on UML properties or on UML slots. Similar stereotypes can reference values in other data sources such as databases.

Furthermore, the UML-based product model can also refer to external computations. This was shown with the representation of a Matlab-specific function as a UML operation. The settings which are required to automatically call an external Matlab function and to return the result were described as properties of a Matlab function-specific stereotype for UML operations. The input and return parameters were mapped accordingly into UML

operation parameters. Similarly, references to other external functions can be represented in UML through UML operations with appropriate stereotypes.

Mathematical dependencies between UML properties were described as algebraic equations. Chapter 6 presented the description of algebraic equations through UML opaque expressions which were owned by UML constraints. The set of equations was resolved through a solution path generator algorithm and a mathematical toolbox. The interpretation of UML opaque expressions depends on their language. Further types of equations can thus be described. As an example, the resolution of equations involving matrices was demonstrated.

Chapter 7

UML model for centralized workflows

As presented in the previous Chapters 4 to 6, application-specific geometric and dynamic system models as well as spreadsheet data can be mapped into a common UML-based product model. In order to guarantee data consistency, dependencies between application-specific models can be defined within the common UML model. Section 7.1 presents the different methods to define these interdisciplinary dependencies. As a result, the automatic update or generation of new application-specific models, based on changes within the central UML model, allows an efficient and consistent evaluation of different product configurations. Section 7.2 shows the process allowing to customize the central UML-based product model and to propagate automatically, as in UML-based software engineering, changes from the central UML model to application-specific models. Moreover, each transaction with the central model can be automated through a programming interface in Java and design processes can be graphically described through UML activity diagrams. Section 7.3 introduces the Java application programming interface and the executability of UML activity diagrams. Section 7.4 describes the frameworks that have been used to implement the translators between the UML-based central product model and each application-specific model.

7.1 UML-based modeling of dependencies

The process steps allowing to achieve a consistent UML-based product model are summarized graphically in Fig. 7.1. The first step consists of importing application-specific models into a UML model. In a second step, dependencies within the UML-based product model are established. In a third step, the dependencies are resolved. In a fourth step, the consistent UML-based product information is exported back to the application-specific models either by generating new models or by updating existing ones. This

Section presents the various possibilities of linking multidisciplinary product data in a common UML model. They include the constraints, generalization relationships and the superposition of stereotypes.

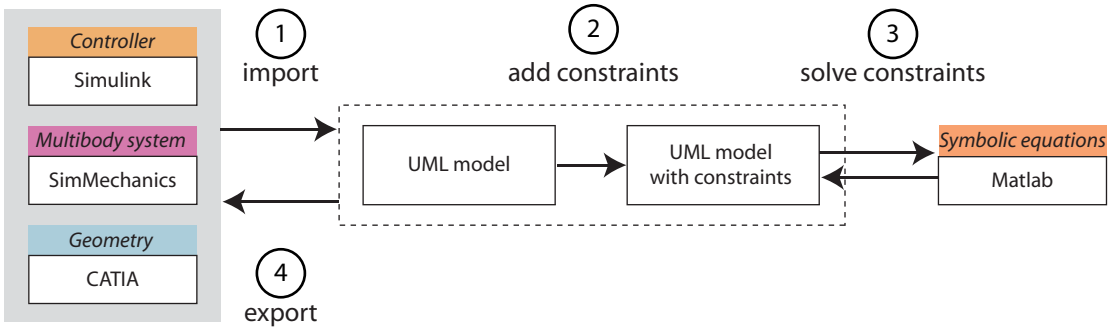

Figure 7.1: Typical process to achieve data consistency within a UML model

Dependencies between UML properties can be defined through UML constraints as presented in Section 6.3. In the case of the slider-crank mechanism, several CATIA measures must match SimMechanics attributes. As presented in Section 4.1.2, the center of gravity position of the rod is for example measured by CATIA under the term inertia center (Fig. 7.4 left) and is saved in the corresponding «$catiaCG$» centerOfGravity property of the «$catiaPart$» Rod class (Fig. 7.3). As shown in Section 5.2.2, the SimMechanics-specific center of gravity position is specified by the origin position vector attribute of the CG coordinate system belonging to the rod body block (Fig. 7.4 right). It is described in UML as originPosition attribute of the related «$simMechCG$» CG port of the «$simMechBody$» Body2CS class (Fig. 7.3).

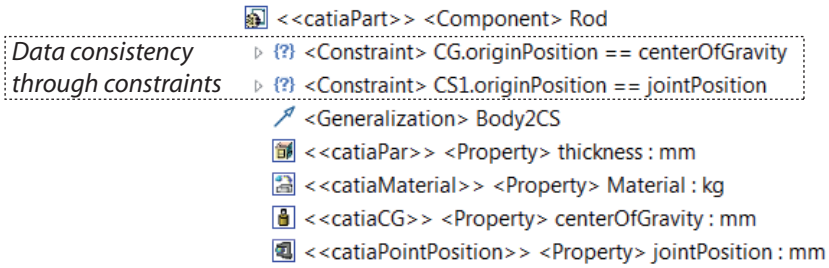

Figure 7.2: UML constraints to describe dependencies

Both CATIA and SimMechanics center of gravity positions are relative to an identical inertial coordinate system and thus need to be equal. A UML constraint describes this relation in an opaque expression with the body $CG.originPosition == centerOfGravity$ as displayed in Fig. 7.2. As presented in Section 5.2.4, the SimMechanics rodInstance is composed of the CS1 and CS2 coordinate systems. A similar constraint is needed to set

the position of the CS1 coordinate system of the SimMechanics rodInstance body block (Section 5.2.4) equal to a point position measured by CATIA. The definition of CS2 coordinate system of the SimMechanics rodInstance body block does not require a UML constraint as it is already defined in relation to the CS1 coordinate system in the SimMechanics model.

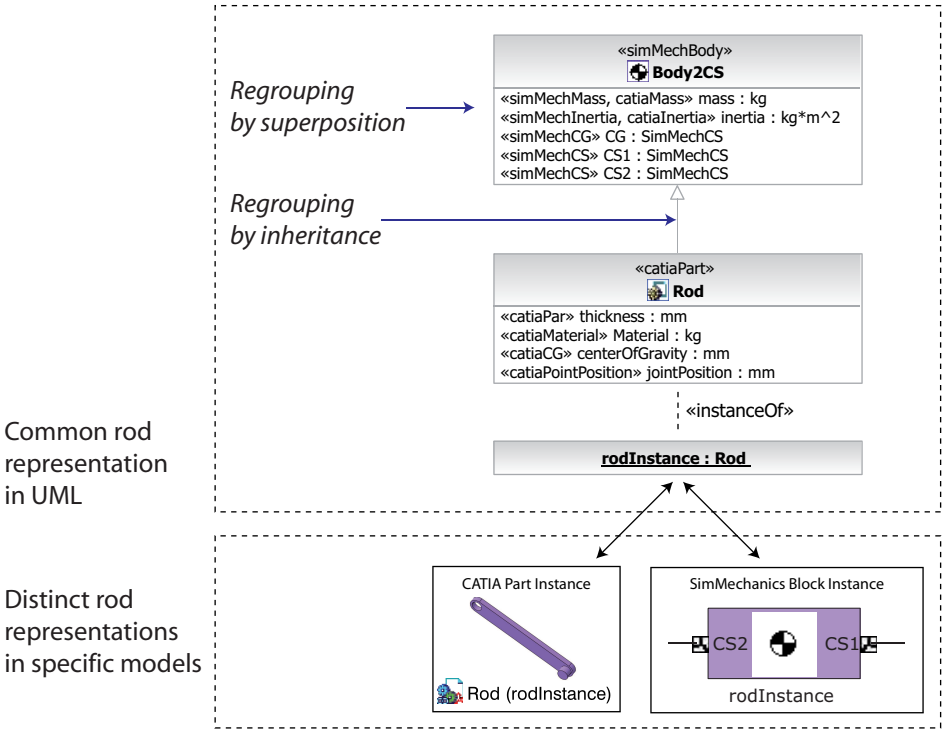

Figure 7.3: Regrouping application-specific properties by inheritance and superposition of stereotypes. Example: CATIA and SimMechanics features common to a rod instance

UML generalization relationships can describe dependencies between application-specific model properties. A UML instance can regroup different properties by having multiple inherited classifiers. The crank, rod and slider parts from the slider mechanism example of Chapters 4 and 5 are represented in UML by the crankInstance, rodInstance and sliderInstance instances. All instances share the same SimMechanics-specific Body2CS class as in Fig. 5.10 but belong to different CATIA-specific classes as each part has a different geometry. A UML generalization relationship is therefore appropriate between the specialized CATIA-specific classes and the common SimMechanics class. The *«catiaPart»* Rod class has for example a generalization relationship with the *«simMechBody»* Body2CS class as displayed in Fig. 7.3. The UML crank, rod, and slider instances are instances of their CATIA-specific class and through the generalization relationship also instances of the common SimMechanics specific class. The instances thus unite the separated CATIA- and SimMechanics-specific properties and can

be translated either into a CATIA product model resulting in part instances or into a SimMechanics multibody system model resulting in block instances (Fig. 7.3).

The same UML property can be relevant for several domains and thus be tagged with several domain-specific stereotypes. In the slider-crank mechanism example, the SimMechanics-specific multibody model requires the inertial part properties of the CATIA-specific geometric model as displayed in Fig. 7.4. The SimMechanics body blocks are specified by their mass attribute, so the corresponding «$simMechBody$» Body2CS class owns a mass property tagged with a «$simMechMass$» stereotype (Fig. 7.3). The correct mass values are imported by CATIA, so each of the CATIA-specific UML classes representing the CATIA parts of the slider-crank mechanism has a mass property tagged with a «$catiaMass$» stereotype (Fig. 7.3). As the mass property is common to all CATIA-specific classes, it can be placed in the generalized Body2CS class which already owns a semantically equivalent mass property. The Body2CS class can either own two mass properties with respectively different names and stereotypes or bind the semantically equivalent properties in one common property tagged with both stereotypes (Fig. 7.3). The latter approach automatically guarantees data consistency as the CATIA and SimMechanics mass values are identical in the UML model and as a consequence also concur in the corresponding CATIA and SimMechanics models (Fig. 7.4).

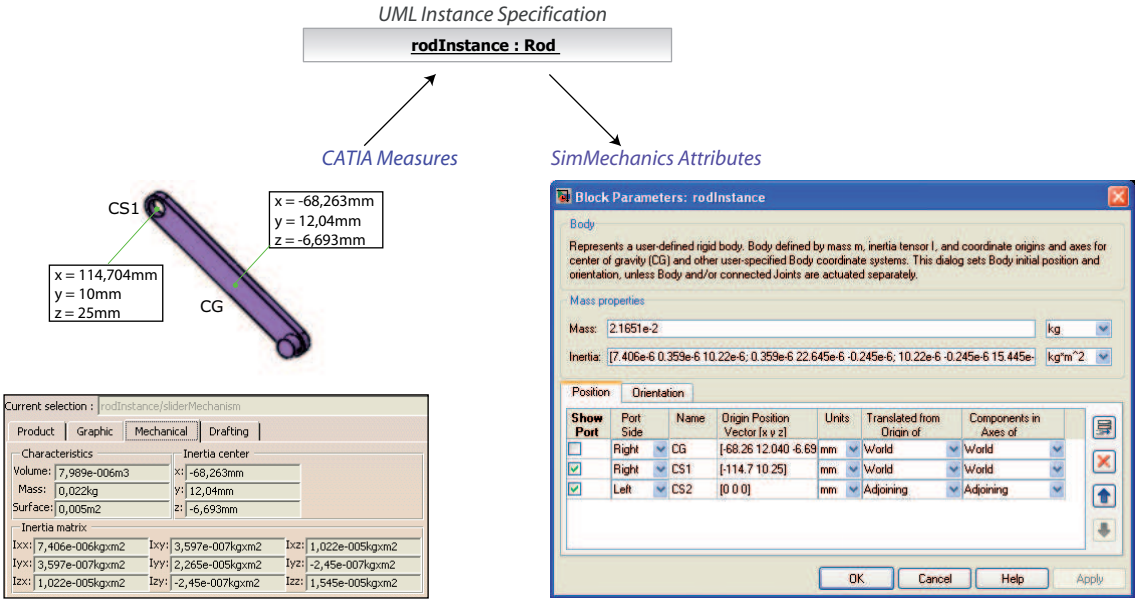

Figure 7.4: Dependencies between the CATIA and SimMechanics rod instances

7.2 UML-based model customization

The central UML-based product model includes the product information which is shared by different disciplines and applications. It also represents application-specific modeling elements in order to automatically translate its application-specific information into application-specific models. Through the translation principles presented in Chapters 4 to 6, centrally defined changes to the UML-based product model can be propagated automatically to application-specific models. This allows an efficient generation or update of application-specific models in order to achieve consistent simulations.

The customizability of a UML-based product model is shown with the example of the slider-crank mechanism. The showcased central UML model is composed of representations of the application-specific CATIA, Simulink and SimMechanics models as depicted in step one of Fig. 7.1. The inter-model dependencies are specified in the UML model as described in Section 7.1. The automatic translation steps in order to achieve a consistent simulation of the controllable slider-crank mechanism are shown in Fig. 7.5. In this scenario, the length of the crank part is shortened from 50mm to 30mm and the derivative gain of the PID controller is increased from 1.5 to 3.

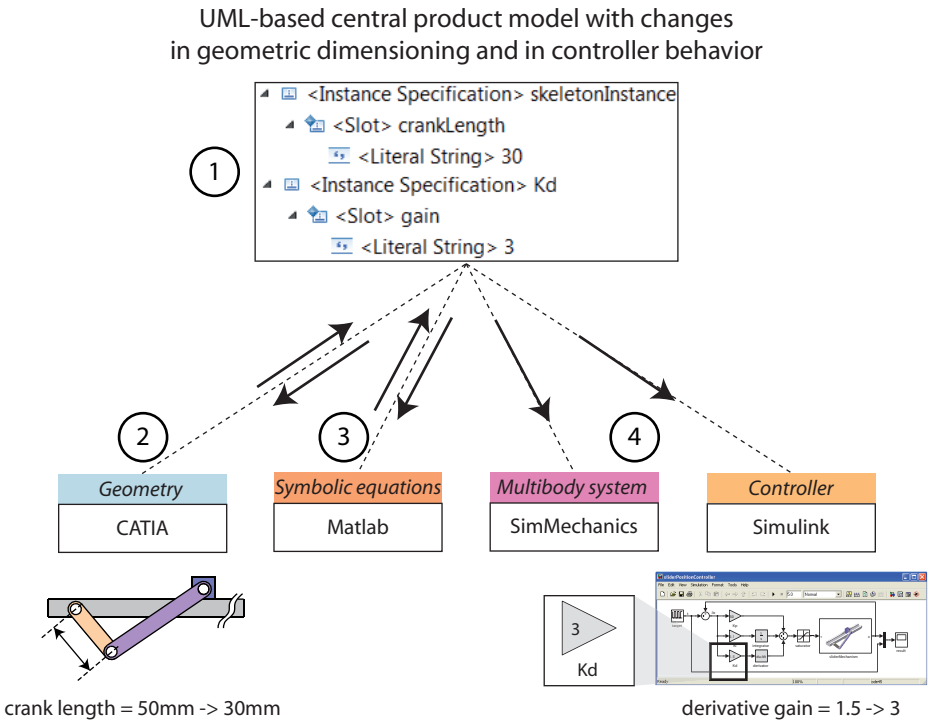

Figure 7.5: Translation steps for a consistent simulation of the controllable slider-crank mechanism based on a central and customizable UML-based product model

The changes are first described centrally in the UML-based product model (Step one in Fig. 7.5). The change of the crank length has an impact on the corresponding geometric and multibody system models as well as on the simulation of the controlled mechanism motion. In the case of the slider-crank mechanism, the geometry of the mechanism can be adapted according to the parameter values of the UML skeleton part instance as presented in Section 4.1.6. So the crank length is a property of the skeleton part instance (Fig. 7.5). The geometric information within the UML-based product model is first translated into the application-specific geometric model. As a result, a new geometric assembly model is generated and the new position and inertial properties of the parts are measured by the geometric application and sent back to the UML model (Step two in Fig. 7.5). CATIA was used as geometric application. The updated UML properties representing the geometric part measures are then set equal to the corresponding UML properties representing the multibody system part attributes through the resolution of symbolic equations by Matlab (Step three in Fig. 7.5). The UML properties specific to the multibody system are then consistent with the UML properties specific to the new geometric model. The change of the derivative gain only has an impact on the controller model. Simulink and SimMechanics are respectively used as applications for dynamic system and multibody system models. Lastly, the UML model generates Simulink and SimMechanics models for a consistent simulation of the controlled slider-crank-mechanism motion (Step four in Fig. 7.5).

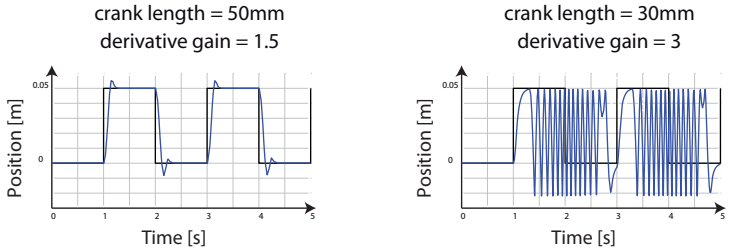

Figure 7.6: Simulation of two controllable slider-crank mechanism configurations

The simulation of the controllable slider-crank-mechanism was presented in Section 5.1.5. The results of the simulation are presented in Fig. 7.6 for the original scenario without change, whereby the crank length and the derivative gain are respectively equal to 50mm and 1.5, and for the new scenario, whereby the crank length is reduced to 30mm and the derivative gain increased to 3. The simulation result presents the position of the slider in meters over time in seconds. The target slider position is colored black and the actual slider position blue. The target position follows a staircase pattern. In the first case, the slider reaches the target position with a slight overshooting at each target position adjustment. In the second case, the slider approaches the target position more slowly

as the derivation gain has been increased. However, the slider misses the target position by little. It is physically impossible for the slider to reach the target position due to the reduced crank length. Consequently, the slider is pushed back and forth on the base rail and cannot reach the target position in the second scenario.

7.3 Automated workflows

The UML-based product model, as presented in this thesis, is based on an application-specific integration approach and is independent of any product design process. As a result, the UML-based product model can be used in different product design processes which may include frameworks for model-based or knowledge-based engineering.

UML models can be created or customized through Java programs. A Java application programming interface (API) for UML models is provided by the Eclipse UML2 Project[1]. The Java API was designed, and mostly generated, based on a UML metamodel definition within the Eclipse Modeling Framework[2]. UML models which are created by the API of the Eclipse UML2 Project are saved in XMI and thus importable by professional UML editors for further use, such as for graphical display.

UML models can include large quantities of product information. Repetitive manual manipulations of UML models in tree editors or in graphical diagrams are therefore often error-prone and time-consuming. These tasks can be programmed through the Java API of the UML model. Moreover, the translators between the UML model and the application-specific models can also be invoked through a Java API. As a result, a multidisciplinary design process consisting of several design evaluations and iterations can be automated. In the case of the slider-crank mechanism, the minimum required power supply for a specific slider configuration can for example be automatically determined iteratively through the generation and subsequent evaluation of different consistent slider-crank mechanism configurations.

Furthermore, the UML supports the graphical description of design processes through activity diagrams. As described in Section 3.3.2, UML activity diagrams are composed of nodes and edges to represent various process steps. Activity diagrams represent a more intuitive representation of design processes than Java programs. As a consequence, a design process described as an activity diagram can be shared and understood by more parties. In this thesis, the executability of activity diagrams was implemented by linking actions within an activity diagram with Java methods. Figure 7.7 for example presents

[1] Model Development Tools - UML2, `http://www.eclipse.org/modeling/mdt/uml2/`
[2] Eclipse Modeling Framework Project (EMF), `http://www.eclipse.org/modeling/emf/`

the reference of a UML setCrankLength action to a corresponding Java method whose invocation changes the crank length in the UML model. The activity diagram is executed by invoking Java methods corresponding to UML actions. This was implemented using the Java reflection package[3] which is for example also used by debuggers.

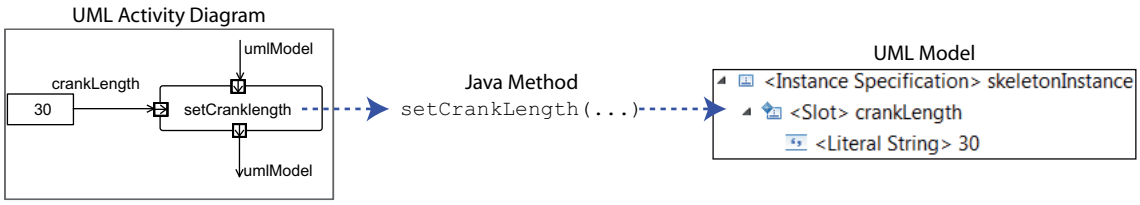

Figure 7.7: Executable UML activity diagrams by linking UML actions with Java methods

Typical design criteria, steps, decisions or rules can be described in Java methods and represented graphically through UML activity nodes. A UML-based product model can thus be automatically customized by an executable UML representation of a product design process (Fig. 7.7). Chapter 8 presents case studies including both Java programs and UML activity diagrams for an automatic generation and customization of UML-based product models.

7.4 Software implementation

The Application Programming Interface (API) of applications was used to create or parse application-specific models. The Visual Basic Script (VBS) API of CATIA and the Visual Basic (VB) API of SolidWorks were used. The translation of a UML model into a CATIA- or SolidWorks-specific geometric model was implemented by generating and executing a corresponding VBS or VB script. The translation of a geometric model into UML was implemented by using temporary XML files. The Eclipse Modeling Framework was used to specify the XML schema of temporary XML files and to automatically generate corresponding Java APIs for an easy reading of the XML files. Based on the geometric information within the temporary XML files, corresponding UML models were created. Next to the application-specific APIs of geometric models, Java APIs for Matlab and Excel were provided by the respective JMatlink[4] and Apache POI[5] projects.

Simulink and SimMechanics models were directly accessed through their text files in ASCII format. Using regular expressions, the Simulink/SimMechanics information

[3]Package java.lang.reflect, `http://java.sun.com/javase/6/docs/api/java/lang/reflect/package-summary.html`
[4]JMatLink, `http://jmatlink.sourceforge.net/`
[5]Apache POI, `http://poi.apache.org/`

was extracted from the model text files and corresponding UML models were programmatically generated based on the Java API of the Eclipse UML2 project. Simulink/SimMechanics model text files were directly generated from the UML-based representation of Simulink/SimMechanics-specific information. Similarly, VRML text files were also directly generated based on the VRML-specific information within the UML models.

The Design Compiler 43v2[6] software was developed to support among other features UML-based product design. It is built upon the open-source Eclipse Platform[7] which supports the design of integrated development environments (IDE). The Eclipse Platform provides the core functionality of integrated development environments and is designed to be extended by software modules which are called plugins. Professional plugins have for example been developed which offer IDEs for Java[8] or C/C++[9] programming. Other plugins offer UML editors such as Topcased[10] or the Eclipse Graphical Modeling[11] (GMF) framework.

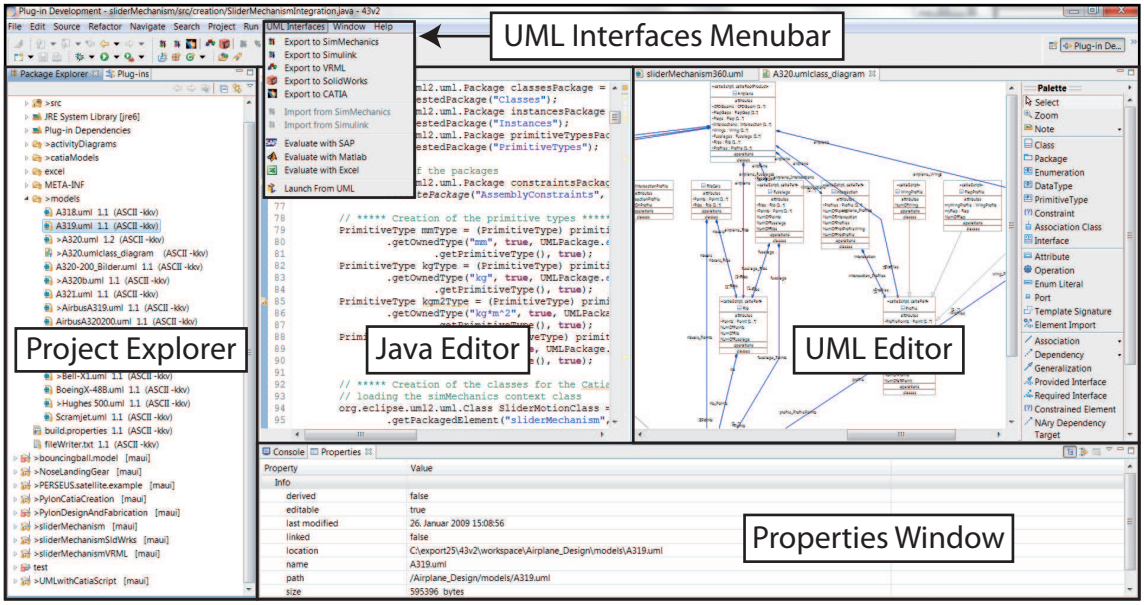

Figure 7.8: Design Compiler 43v2 based on the Eclipse Platform

Figure 7.8 presents the Design Compiler 43v2 software based on the Eclipse platform. The software allowing to translate application-specific models into UML and vice versa was equally bundled into Eclipse plugins. Similarly, the software needed to execute UML activity diagrams was also bundled into a plugin. The UML translator plugins

[6]Design Compiler 43v2, `http://www.iils.de/`
[7]Eclipse Platform, `http://www.eclipse.org/platform/`
[8]Eclipse Java development tools (JDT), `http://www.eclipse.org/jdt/`
[9]Eclipse C/C++ Development Tooling (CDT), `http://www.eclipse.org/cdt/`
[10]Topcased, `http://www.topcased.org/]`
[11]Eclipse Graphical Modeling Framework (GMF), `www.eclipse.org/gmf/`

for UML-based product design have not changed the graphical user interface of Eclipse apart from adding menubar, toolbar and popup menu actions. Software modules required for UML-based product design are regrouped in one software environment through the Eclipse Platform. The Design Compiler 43v2 for example includes, next to the UML translator plugins, a Java IDE and UML editors. The modular architecture of the Eclipse Platform facilitates the integration of further features.

7.5 Summary

Chapter 7 has presented centralized workflows between heterogeneous application-specific models through the use of a UML-based central product model. The definition and evaluation of dependencies within the UML-based central product model is required in order to achieve data consistency between various application-specific models. Inter-model dependencies were represented in UML by the definition of constraints, the regrouping of properties by inheritance and superposition of stereotypes.

Furthermore, the UML-based central product model supported model customization. Changes were defined centrally in the UML model and propagated automatically to the dependent application-specific models. The automated customization of application-specific models is more efficient than the manual update of separate application-specific models. The approach was shown for the evaluation of a new slider-crank mechanism configuration.

The UML-based product model is based on an application-centric integration approach and is independent of any product design process. As a result, the UML-based product model may be used within different product design processes. The Java API of the UML model allows other frameworks, such as for model-based or knowledge-based engineering, to access the UML-based product model. Furthermore, UML activity diagrams can graphically represent product design processes. Their executability was made possible by linking activity nodes with Java methods. Product design processes can thus be represented in UML activity diagrams and be executed to generate or modify UML-based product models and translate them into application-specific models.

The software for the translation of UML models into application-specific models and vice versa, as well as the software for the executability of UML activity diagrams, were bundled into Eclipse plugins. The Design Compiler 43v2 software includes among others these plugins as well as freely available plugins for Java programming and UML model editing. As the Design Compiler 43v2 software is based on the Eclipse platform, it can easily integrate further features.

Chapter 8

Test cases

This chapter reviews projects which were undertaken in partnership with academia and industry to highlight the use of a UML-based central product model for the automated generation of consistent model configurations. The test cases include models for the evaluation of different cabin pressure control systems (Section 8.1), the generation of customizable conveyor systems (Section 8.2), the automatic evaluation of different satellite configurations (Section 8.3) and the generation of aircraft geometry (Section 8.4).

8.1 Evaluation of cabin pressure control systems

This Section presents the evaluation of cabin pressure control systems based on an integrated modular avionics (IMA) architecture. The project was undertaken in partnership with the Institut für Luftfahrtsysteme[1] of the University of Stuttgart. IMA includes many engineering aspects ranging from hardware to software. It is therefore hard to integrate the numerous IMA aspects into a single model in order to achieve an evaluation of an IMA architecture.

A cabin pressure control system is responsible for ensuring human-friendly pressure conditions within an aircraft cabin. In this test case, the cabin pressure control system was designed according to an IMA architecture [125]. Avionic systems were traditionally designed as a federated architecture of computing resources dedicated to specific functions. In IMA, several functions share the same computing resource so the number of required on-board computers is reduced. As a result, the IMA paradigm is intended to reduce the mass, volume and power consumption of avionic architectures. IMA is a multidisciplinary task as it consists of ensuring the reliable functionality of many systems while at the same time minimizing the mass, volume and cost of the complete avionic ar-

[1] Institut für Luftfahrtsysteme (ILS), `http://www.ils.uni-stuttgart.de/`

chitecture. Due to the high diversity of computing and communication resources as well as the many topological configuration possibilities, the evaluation of many different IMA architectures is required in order to reach an optimal configuration.

Components within an IMA architecture are intended to be easily replaceable and configurable. IMA architectures are therefore composed of commercial off-the-shelf (COTS) components in order to be scalable and adaptable for several different aircraft types. An IMA architecture is composed of processing units and different types of links for inter-module communication. An example of a module is the Core Processing Input/Output Module (CPIOM) to execute specific avionic functions. They provide computing capabilities for various applications and replace the traditional black box concept.

Figure 8.1 presents two different cabin pressure control system architectures which are not aircraft type-specific. They consist for example of CPIOMs, outflow valve control and sensor modules (OCSM), outflow valves, pressure sensors and outflow relief valve dumps (ORVD). These processing modules are interconnected through different types of buses such as ARINC 429, CAN, RS422 and AFDX.

Selected aircraft sections are reserved for the safe positioning of processing modules within the fuselage. The selected sections provide access to power and enable the diffusion of heat. Furthermore, they are easily accessible to maintenance personnel. Nevertheless, a cabin pressure control system can vary in the choice of processing modules and in the type of communication buses, as depicted in Fig. 8.1. Architecture characteristics such as mass, volume and cost need to be evaluated in order to objectively compare different architecture configurations.

The components of the cabin pressure system were represented as UML classes and instances. Properties which were common to several classes were represented in a common abstract class. The classification hierarchy of the cabin pressure system is represented in the class diagram of Fig. 8.2. The directed lines within the class diagram show inheritance relationships.

All components have a mass and a price. These properties were therefore placed in the highest class within the class inheritance hierarchy, namely in the Component class. As a consequence, the mass and price properties were inherited automatically by all lower classes. The properties common to all computing resources, such as position and volume, were described in the Unit class and similarly the properties common to all connections, such as length and mass per length, in the Connection class. The volume computation of the units assumed that the units were either of rectangular or cylindrical shape. The specific unit and connection types which appeared in the cabin pressure system were situated at the lowest level within the class hierarchy.

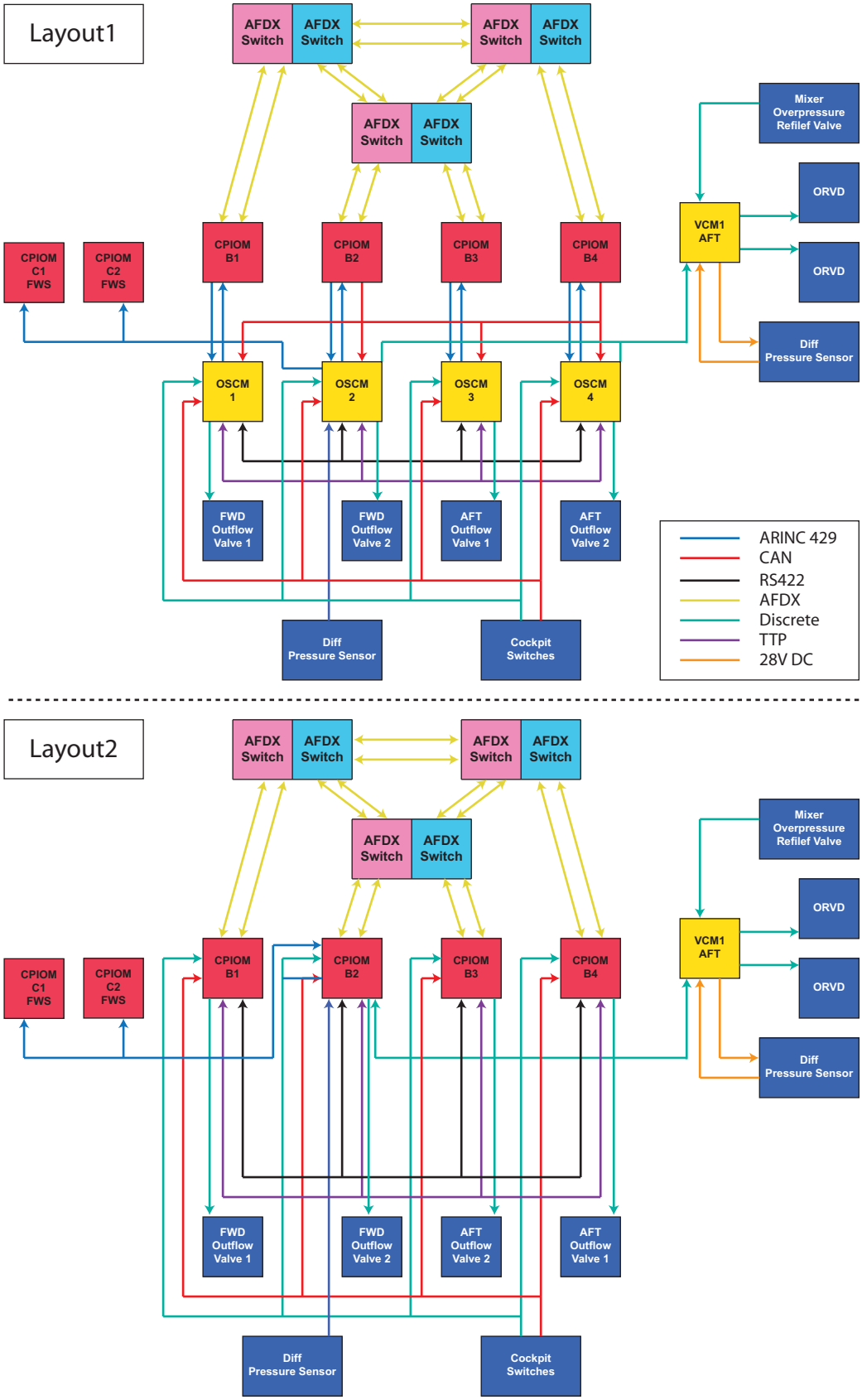

Figure 8.1: Examples of different cabin pressure control system architectures

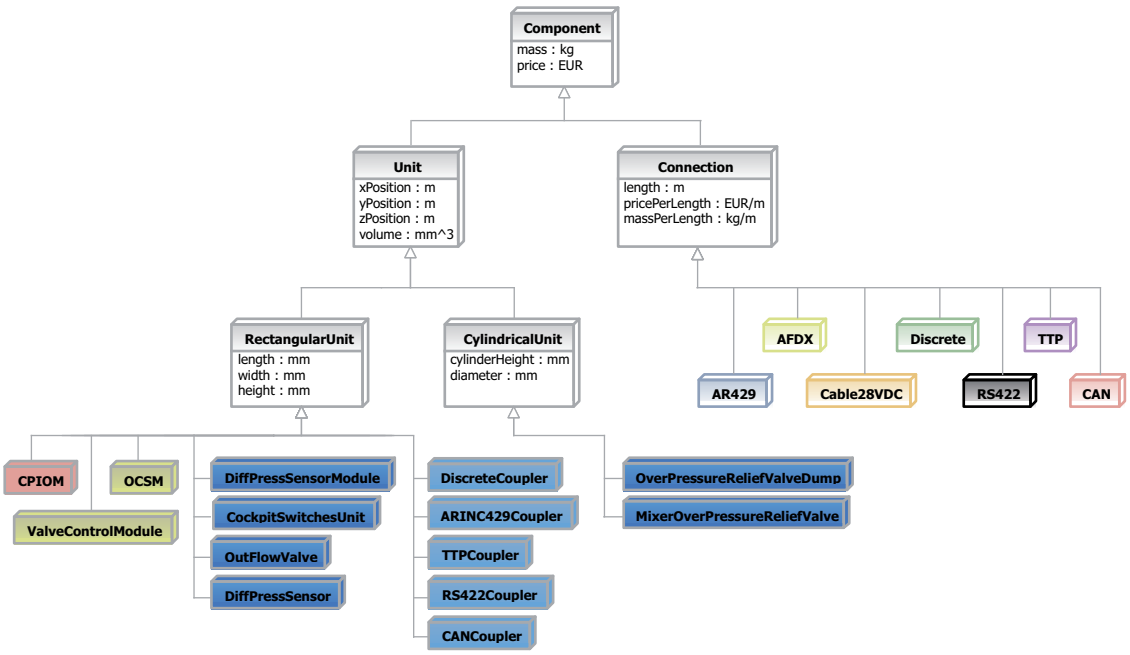

Figure 8.2: UML class diagram showing the generalization relationships within the cabin pressure system components

Each computing resource has input/output connection possibilities. A CPIOM can for example have up to 4 connections of type AFDX and up to 10 connections of type ARINC 429. Each connection possibility of a certain type was described in UML as a property. The CPIOM class for example had properties named AFDXConnections and ARINC429Connections. These properties were represented as UML associations in the class diagram of Fig. 8.3. The maximum number of connection possibilities of a certain type was represented by the multiplicity of their respective properties and was displayed graphically next to the name of the associations.

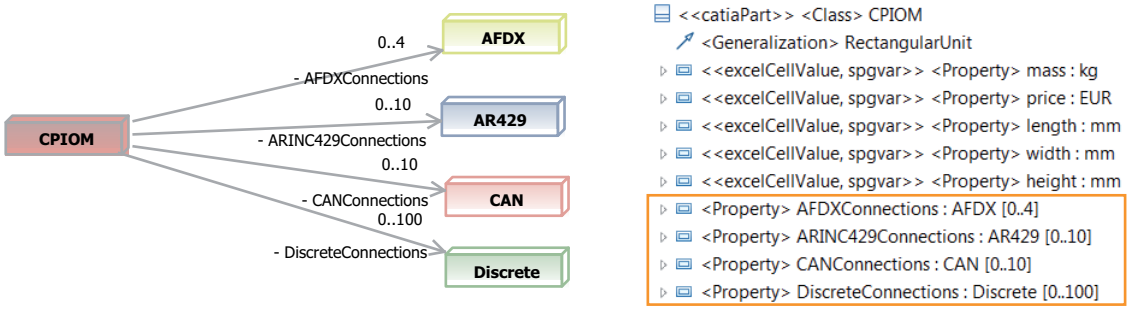

Figure 8.3: UML class diagram showing the associations of the CPIOM class

The UML instances describing a concrete cabin pressure system layout could be represented graphically in a UML object diagram. However, as the UML instances were

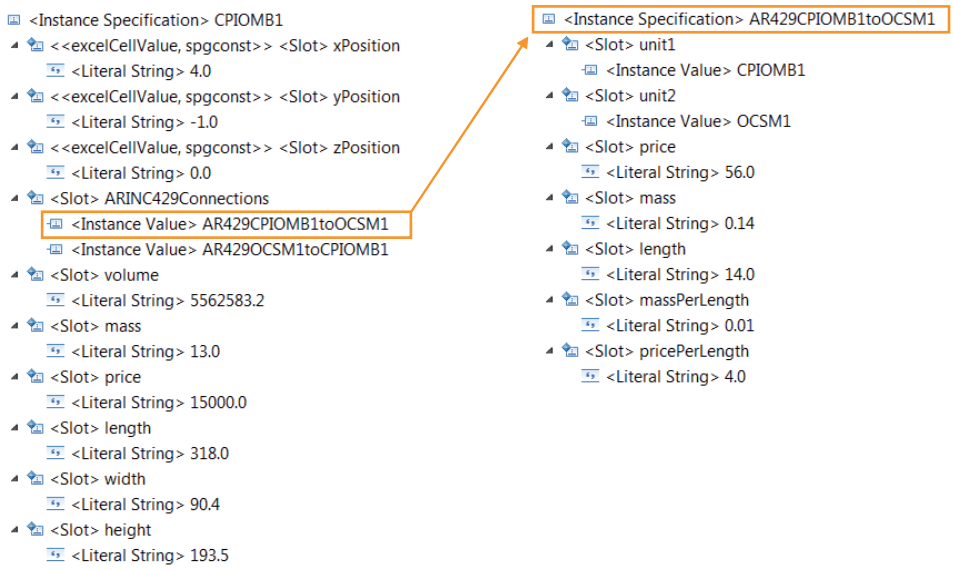

Figure 8.4: Example of CPIOM and ARINC429 instances

very numerous, a resulting object diagram would not have been easily comprehensible. Therefore, the values of instance slots which were most often of interest were represented in tree views. Figure 8.4 for example shows the attribute values of a CPIOM and an ARINC429 instance. The CPIOM instance has an instance value referring to the ARINC429 instance under its ARINC429Connections slot.

Many values related to the cabin pressure system are available in Excel spreadsheets. The Excel values which applied to all UML class instances were attached to the corresponding class properties through Excel-specific stereotypes. Similarly, the values which only applied to specific class instances were added onto their respective instance slots. Figure 8.5 examplarily shows the application of an Excel-specific stereotype to the xPo-

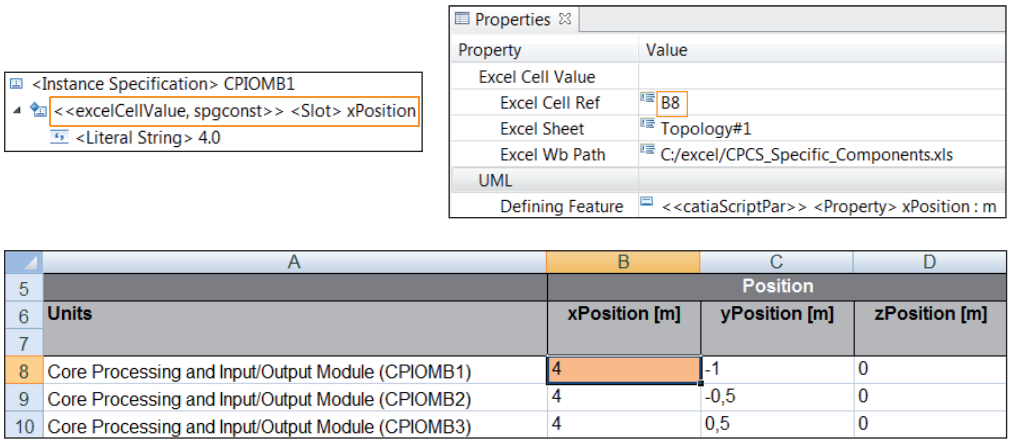

Figure 8.5: UML-based representation of Excel cell values

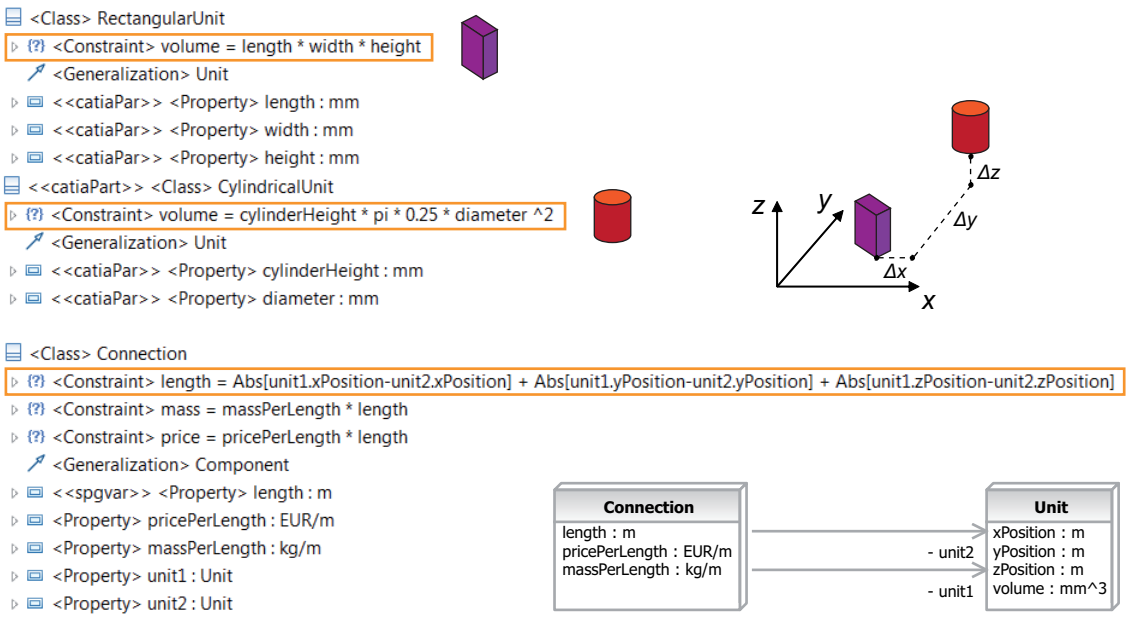

Figure 8.6: UML constraints to compute component volumes and connection lengths

sition slot of a CPIOM instance. The stereotype properties refer to a cell value within a specific spreadsheet inside an Excel document. Similarly, other attributes such as the mass and the price of components were imported from Excel.

The volume of every component needs to be computed in order to evaluate the overall volume of a cabin pressure system configuration. The volume of a component only depends on its own dimensions. The symbolic equations to compute the volume of rectangular and cylindrical units were described in UML through UML constraints of the UML RectangularUnit and CylindricalUnit classes (Fig. 8.6) respectively. Furthermore, the length of each connection was computed as it had an impact on the mass and price of a cabin pressure system configuration. The length of each connection depended on the position of the connected components and was assumed to follow a rectilinear layout. The corresponding symbolic equation was described in UML through a UML constraint of the UML Connection class (Fig. 8.6). It was assumed that the mass and the price of a connection were proportional to the connection length. As a consequence, the UML Connection class had two UML constraints to compute respectively the connection mass and the price based on its length.

The UML model of the cabin pressure system consisted of too many elements to be built from scratch manually. The UML model was therefore generated by the execution of a Java program. Recurrent design steps to establish the UML model of a cabin pressure system were described in Java methods. The Java program referred to the predefined methods to generate a UML model composed of classes, instances and constraints.

The Java program also applied stereotypes onto the UML elements to allow an import of Excel-specific spreadsheet data into UML as well as an export of the UML-based geometric information to a CATIA-specific geometric model.

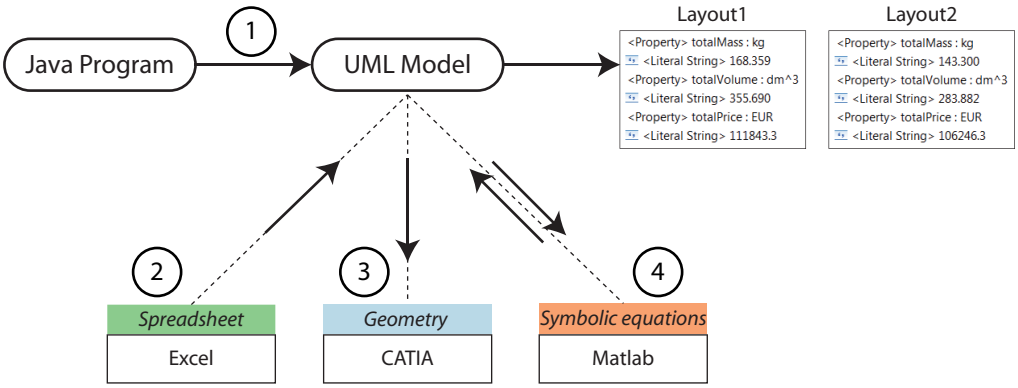

Figure 8.7: Process to efficiently generate and evaluate different cabin pressure system configurations

Figure 8.7 presents the steps leading to the evaluation of a UML-based representation of a cabin pressure control system. The easily adaptable Java program was used to create UML representations of different cabin pressure system configurations (Step 1 in Fig. 8.7). The generated UML models then imported the Excel-specific spreadsheet data of the system components (Step 2 in Fig. 8.7). The spreadsheet data also included the position of the components within the aircraft. The UML model was then exported to a CATIA-specific geometric model to visualize the correct placement of the cabin pressure system components (Step 3 in Fig. 8.7). Figure 8.8 for example presents the placement of cabin pressure system components within an A380 aircraft model. Next, the UML constraints describing the computation of mass, volume and price properties were automatically resolved based upon the solution path generator and the Matlab-specific symbolic toolbox (Step 4 in Fig. 8.7) as described in Section 6.3.

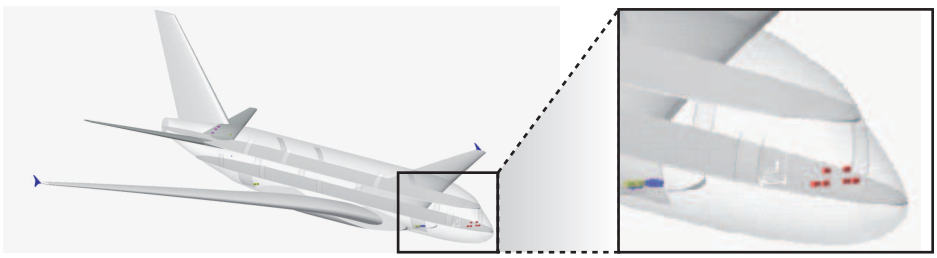

Figure 8.8: Geometric model of cabin pressure system units within an A380

This process was applied to the evaluation of the two different cabin pressure control system architectures of Fig. 8.1. Figure 8.7 presents their evaluation according to their

mass, volume and price. The second layout (Fig. 8.1 bottom) seems more appropriate than the first one (Fig. 8.1 top) as it has a lower mass, volume and price. This is not surprising as the second layout is composed of fewer units. A holistic evaluation of a cabin pressure system would however require the evaluation of many more aspects such as the system reliability.

In general, IMA architectures entail a high degree of multidisciplinarity and multiple potential configuration possibilities. They therefore seem to be the ideal test case for a UML-based central product model. Two versions of a simplified cabin pressure system were used to show the capabilities of a UML-based product model to generate consistent system configurations. The UML-based product model thereby integrated Excel-specific spreadsheet data and CATIA-specific geometric information as well as symbolic equations to compute component values. The approach could be further extended to investigate the integration of other aspects related to IMA architectures.

8.2 Automated design of conveyor system configurations

Conveyor systems are used to transport materials from one place to another. Many kinds of conveying systems are available for various needs in different industries including the automotive, aerospace and packaging sectors. Conveyor systems need to be highly customized for the specific needs of each customer. This Section describes the use of a UML-based product model to customize the design of motor-driven chain conveyors for the painting of automobiles. The geometric models have been provided by one of the leading conveyor suppliers for automobile manufacturing. The UML-based generation of identical geometric models was performed in order to prove the capabilities of a UML-based product model in an industrial context. Furthermore, the customizability of the UML representation of conveyor systems was proven through the generation of several conveyor system versions.

SolidWorks was used as CAD application. The geometric assembly model of the conveyor system was composed of several smaller assemblies called modules. The positioning of the modules occurred through assembly constraints. The top assembly model functioned like a skeleton model by providing planes according to which the embedded modules were placed. Figure 8.9 for example shows the planes of the top assembly model of the conveyor system and the placement of modules based on the coincidence of planes.

The geometric models of modules represented geometric templates which were instantiated and inserted into other assembly models. A geometric module model was therefore represented in UML as a class and a corresponding geometric assembly model

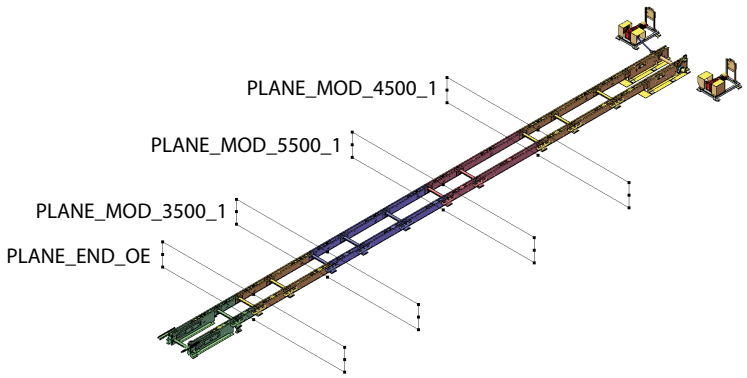

Figure 8.9: Example of a geometric model of the conveyor system. Placement of modules based on the coincidence of planes

instance as a class instance. Furthermore, the modules were categorized in the UML classes InletModule, ChainModule, IntermediateModule, OutletModule and DriverModule according to their common features as in Fig. 8.10. Apart from their diverse functions, the modules differed by having different planes within their geometric models. All modules of type InletModule for example had planes named START_MODULE, Right and FLOOR.

The different inlet modules of varying length were described in separate geometric models. The module length is derivable from their name. Figure 8.11 for example shows the inlet modules of varying length such as 4500mm, 5500mm and 6500mm. As the modules shared the same planes, the corresponding OE_4500, OE_5500 and OE_6500 UML classes all inherited the properties from the common abstract InletModule class. The UML class diagram in Fig. 8.11 displays the common geometric module features which are not explicitly described in the geometry-specific SolidWorks application.

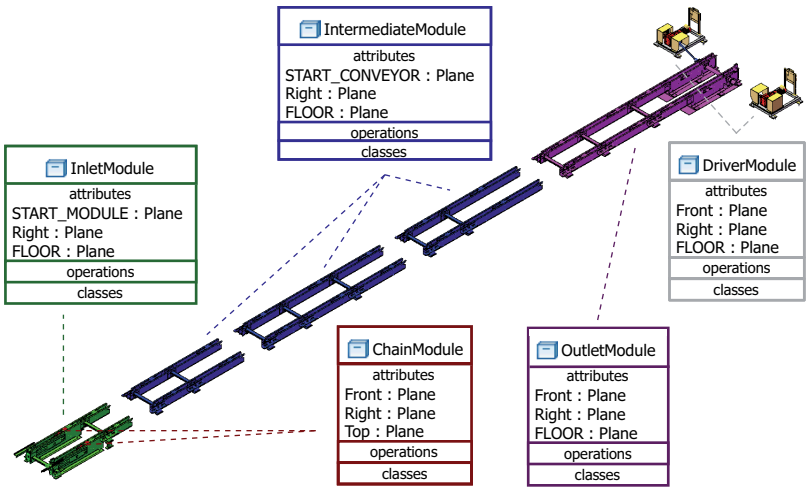

Figure 8.10: Abstract UML classes to classify different module types

Assembly constraints were represented through UML constraints as described in Section 4.2.3 and referred to detailed geometric elements such as planes, sketches, lines and points. A one-to-one mapping of SolidWorks-specific geometric entities into UML allowed to easily recognize SolidWorks-specific information in UML. The low-level geometric entities of modules were represented as UML instances with predefined classifiers as described in Section 4.2.2.

The decomposition of the conveyor system is outlined in the class diagram of Fig. 8.12. The conveyor was represented in UML by the TKFAssembly class, whereby TKF stands for "Trocknerförderer". The geometric model of the complete conveyor system was represented by a SolidWorks-specific assembly model. The TKFAssembly class was therefore tagged with a «$sldWorksRootAsm$» stereotype.

The associations of the TKFAssembly class with other module classes showed the required number of module types. The multiplicity of the inletModule property was for example equal to one while the multiplicity of the shaft, driver and chain modules ranged from one to two. As a consequence, the root assembly model of the conveyor system could only have one inlet module and it required either one or two shaft, driver and chain modules. The geometric elements of the root assembly model were described as UML type-specific properties such as the "planes" and "sketches" properties. Furthermore, the TKFAssembly class had associations with classes which represented non geometry-related conveyor information such as CountryOfmanufacture, MotorModule, MotorType, conveyorSpeed and payloadMass.

The geometric assembly instance and its corresponding UML instance are represented in Fig. 8.13. The assembly instance named "TKF Assembly Instance" is composed of planes, assembly instances and a sketch. The corresponding UML instance is similarly

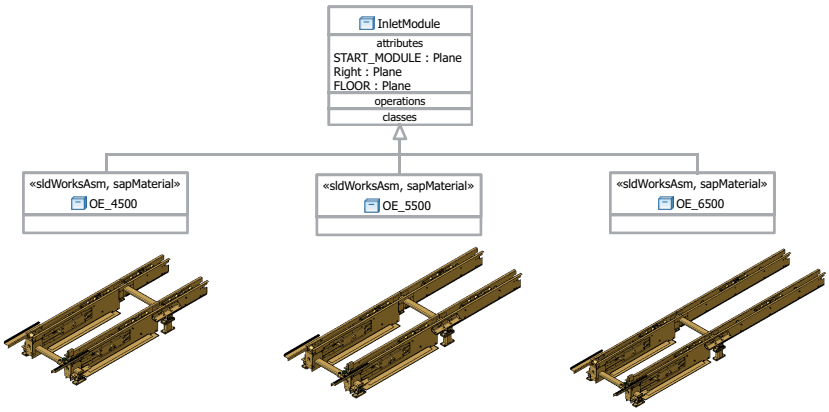

Figure 8.11: Generalization relationships between the module classes and the common abstract InletModule class

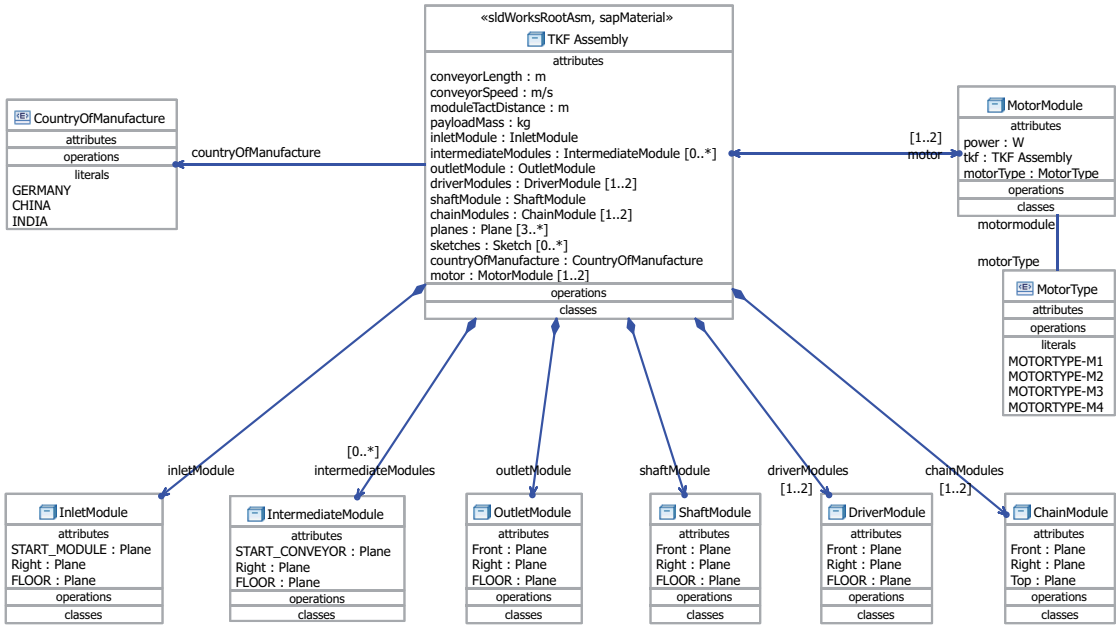

Figure 8.12: UML class diagram showing the main conveyor system classes

composed of slots referring to their respective geometric features, according to the UML TKFAssembly class properties as shown in Fig. 8.12. All references to plane instances were for example stored in the “planes” slot and all references to intermediate modules were similarly stored in the “intermediateModules” slot.

The conveyor could be configured according to several criteria in order to satisfy different requirements. The choice of modules depended on the total conveyor length and on the country-specific module supplier. The conveyor motor depended on the payload mass, conveyor length, conveyor speed and module tact distance. Each new conveyor configuration influenced the geometric model of the conveyor system. The UML-based product model therefore represented the conveyor information which was likely to change. As the UML-based product model also represented SolidWorks-specific modeling elements, changes in the UML model could be automatically translated into a corresponding SolidWorks model. However, the UML-based product model did not include all the detailed geometric information and was thus easier to adapt to changing requirements than the fully detailed SolidWorks-specific geometric model.

Design automation made it possible to ensure consistency between original design requirements and new detailed geometric models. Although the UML model of the conveyor system was not as detailed as the corresponding geometric SolidWorks model, the UML model still included a multitude of modeling elements. Recurrent changes were defined in Java methods whose invocation automatically adapted the UML model. Changes

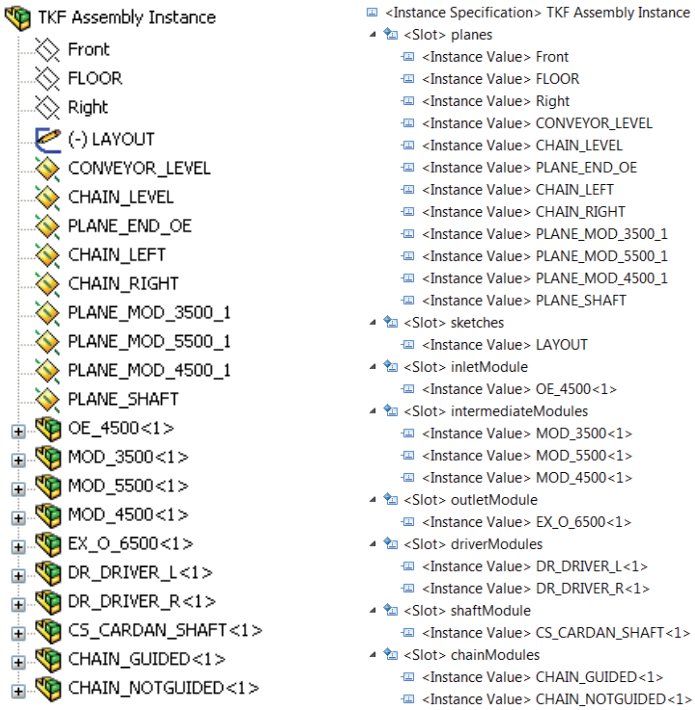

Figure 8.13: Tree view of the SolidWorks assembly model and corresponding UML instance

were consequently implemented faster and with higher reliability than through manual modifications. Figure 8.14 presents an outline of the Java methods which were used to describe typical design steps within the conveyor design process as well as typical design decisions based on variable requirements. The execution of the Java methods within a Java program enabled the generation of new updated UML product models based on which corresponding SolidWorks conveyor models were automatically generated.

As described in Section 7.3, the design steps encoded in Java were depicted graphically in a UML activity diagram (Fig. 8.15). An activity diagram enabled a more transparent view of a design process than a Java program. Variable requirements, design steps and information flows could thus be better illustrated. The activity diagram in Fig. 8.15 presents the design process of the conveyor system. The sequence of design steps and design decisions based on the requirements are visible in the activity diagram. The “chooseMotor” and “addIntermediateModule” actions for example depended on variable design requirements. They were represented through activity input parameters such as “conveyor length” and “conveyor speed”. The “addInletModule” and “addOutletModule” for example represented typical design steps. Activities could also be decomposed into several subactivities. The “addintermediateModule” action was for example described in more detail through a subactivity. The UML model required SolidWorks-specific as well as

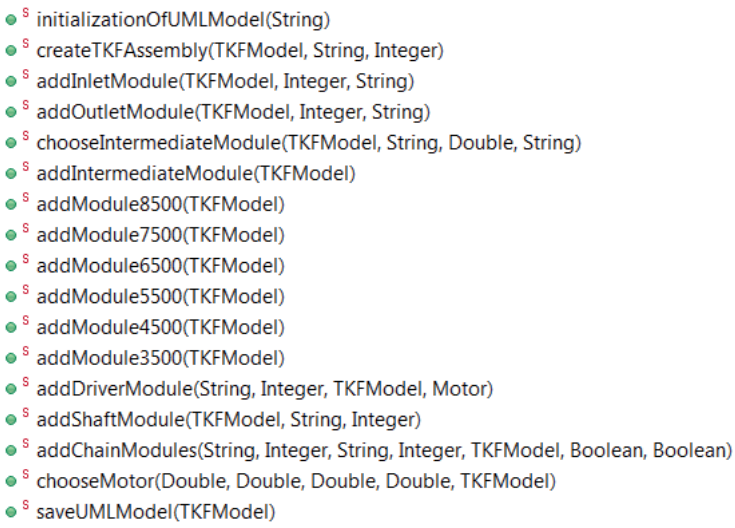

Figure 8.14: Java methods describing recurrent conveyor design decisions and steps

SAP-specific information to respectively describe the geometry and the bill of materials (BOM) of the conveyor system. The SolidWorks-specific and SAP-specific information was represented like the conveyor requirements through activity input parameter nodes respectively colored yellow and blue.

The requirements as well as the SolidWorks- and SAP-specific information could be directly edited within the UML activity parameter nodes. The actions of the activity diagram referred to the Java methods of Fig. 8.14. As described in Section 7.3, the activity diagram could thus be executed similarly to a Java program. The UML activity diagram of the conveyor design process was used to generate different UML models of conveyor systems, based upon which consistent SolidWorks-specific geometric models were automatically produced.

This Section presented the UML-based representation of SolidWorks-specific geometric models related to conveyor systems. The translation of UML into SolidWorks resulted in geometric models identical to the manually edited models commonly used in an industrial context. Furthermore, the customizability of the UML-based conveyor system model was made possible by describing typical design requirements and design steps in a UML activity diagram. The executability of the activity diagram based on underlying Java methods allowed the efficient generation of new conveyor system model configurations.

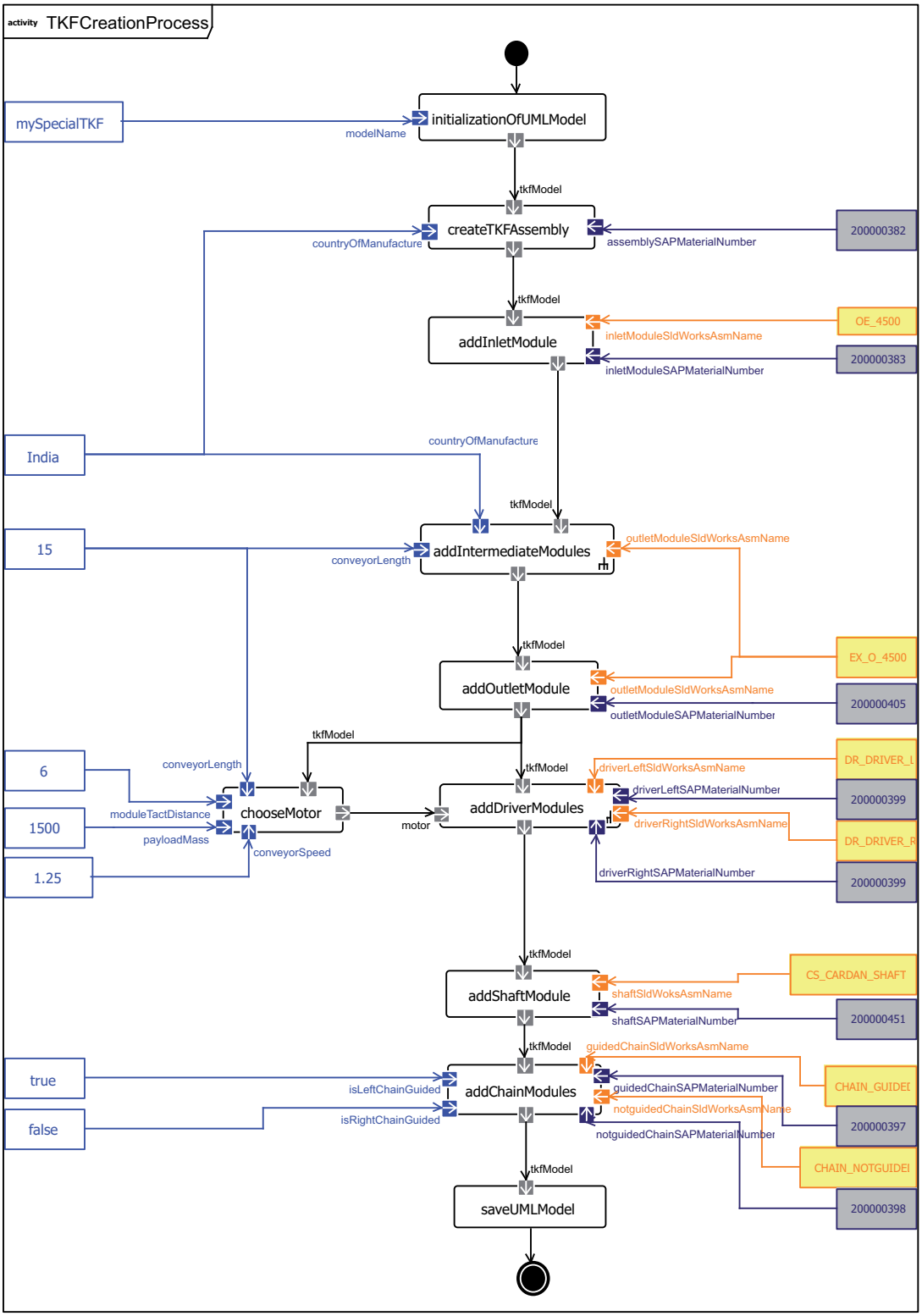

Figure 8.15: Activity diagram of the conveyor design process

8.3 Automated evaluation of satellite configurations

Due to the multidisciplinary nature of satellite design and the mostly proprietary formats of engineering software tools, a multitude of heterogeneous computer models are employed during satellite design. A unified central product model can manage the interdependencies between different isolated models and guarantee data consistency. This Section highlights the UML as central product model to support satellite design [53, 54]. The approach has been applied to the design phase of the Perseus satellite [16] which is part of the Stuttgart Small Satellite Program[2] of the Institute of Space Systems[3] at the University of Stuttgart. This Section presents the UML lightweight extensions necessary to represent geometric features authored in CATIA and control system features authored in Matlab/Simulink in a common UML-based product model. Furthermore, the representation of a design process as an executable UML activity diagram is shown for an iterative design sequence consisting of several CATIA- and Matlab/Simulink-specific evaluations.

The Perseus satellite is equipped with two different electrical propulsion systems. The Perseus mission is intended to accomplish the in-orbit test and validation of new low-cost electric thruster systems during its flight to the moon. Afterwards, the satellite is anticipated to accomplish UV astronomy in the spectral band of 120 nm to 180 nm with an on-board telescope. It is designed with different engineering software tools. The two application-specific models of the satellite considered in this Section are a geometric model authored in CATIA and a dynamic model authored in Simulink. The integration of the application-specific data in the UML-based product model was realized by lightweight extensions in the form of stereotypes.

In Fig. 8.16, the CATIA product model of the pulsed plasma thruster (PPT) and its instance within the larger satellite product model are depicted. The corresponding UML elements are represented below. CATIA parts and products were mapped into UML classes respectively with «$catiaPart$» and «$catiaProduct$» stereotypes. CATIA product and part instances were represented by UML instances. The PPT instance for example has PulsedPlasmaThruster as classifier.

CATIA-specific geometric properties need to be represented in the UML-based product model as they have an influence on the dynamic behavior of the complete satellite. CATIA-specific measures belong to a part or a product and were therefore described in UML through corresponding UML class properties. For the integration of CATIA-specific measures into the UML-based product model, stereotypes were applied to the properties of a related «$catiaPart$» or «$catiaProduct$» class. As an example, the posi-

[2]Stuttgart Small Satellite Program, `www.kleinsatelliten.de`

[3]Institut für Raumfahrtsysteme (IRS), `http://www.irs.uni-stuttgart.de/`

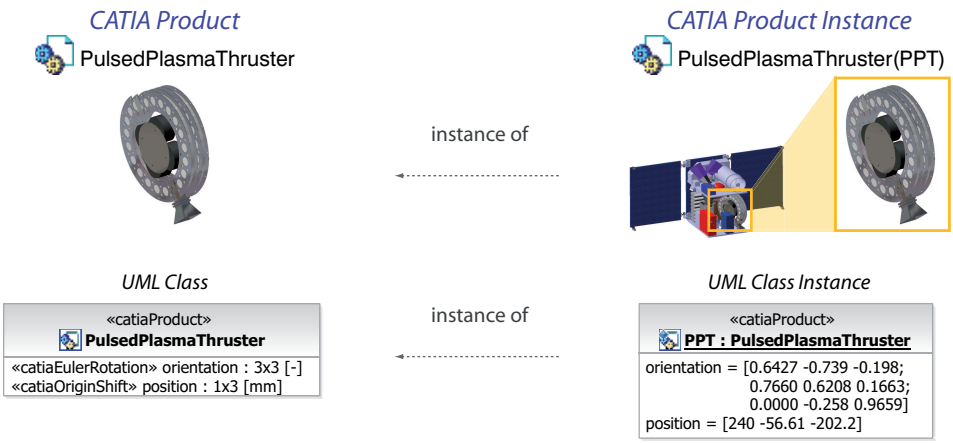

Figure 8.16: Left: CATIA product and corresponding UML class. Right: CATIA product instance and corresponding UML class instance (after Gross et al. [54])

tion and orientation of the thruster were described through properties tagged respectively with «$catiaEulerRotation$» and «$catiaOriginShift$» stereotypes (Fig. 8.16). The values for these properties were retrieved from the CATIA instance and were stored as literal string values in the slots of the UML class instance.

According to the CATIA geometry hierarchy, a product can contain further products or parts. The CATIA-specific composition hierarchy was translated one-to-one into a corresponding UML class composition hierarchy. In Fig. 8.17, the composition of CATIA products and of the related UML classes belonging to the Perseus satellite is depicted. The class representing the top-level product in the composition hierarchy was labeled with a «$catiaRootProduct$» stereotype. The containment relations of the CATIA products were modeled with UML composite aggregations.

CATIA part instances are positioned in an assembly according to assembly constraints. They were translated into UML constraints. CATIA constraints are owned by a product, so the UML constraints were owned by the related «$catiaProduct$» class. According to the type of the assembly constraint, the UML constraint was tagged with a specific stereotype such as «$catiaAngle$» or «$catiaCoincidence$». Whenever necessary, the specific stereotype owned attributes for a complete description of the constraint. A «$catiaAngle$» stereotype for example owned attributes to specify an angle value and an angle sector.

Changes in the UML model, concerning for example the choice of parts or the packaging strategy, were automatically translated into a corresponding CATIA model and vice versa. For large CATIA models, the complete translation of a UML model to generate a new CATIA model or vice versa might have taken too much time. The «$catiaUpdate$» stereotype was therefore applied to UML properties and constraints. Only the CATIA features which corresponded to «$catiaUpdate$» UML elements were then updated. This

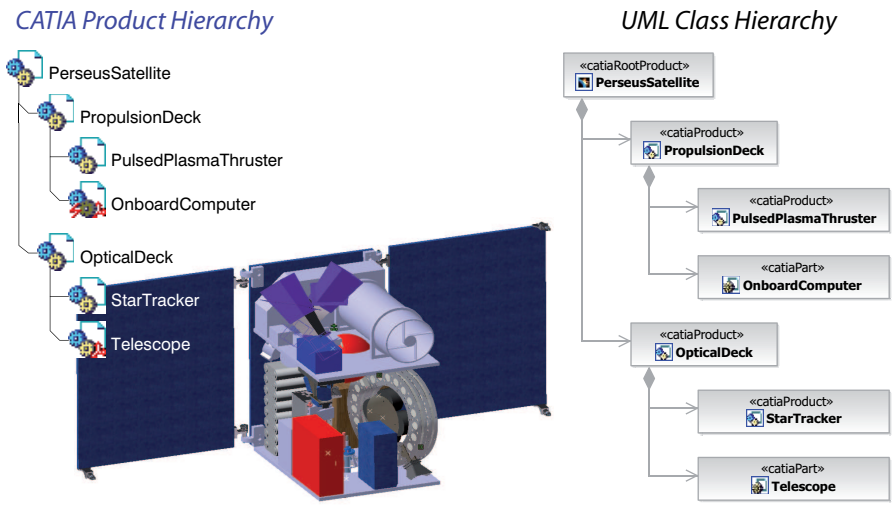

Figure 8.17: Selection of the CATIA model composition hierarchy and corresponding UML class hierarchy (after Gross et al. [54]

allowed a quick update of specific parts within a larger assembly without having to regenerate the complete assembly.

Mathematical equations between properties were described in the UML-based product model through UML constraints between UML properties. Each constraint contained a UML expression representing a symbolic mathematical equation. The orientation and position properties for example of the PulsedPlasmaThruster class in Fig. 8.18 represented CATIA-specific measures which needed to be converted to corresponding Simulink-specific A_PPT_Thr and O_PPT_Thr properties according to another reference frame. The computation of the O_PPT_Thr property for example depended on the orientation and position properties as displayed in the equation colored blue. The equations were computed using the values of the UML instances stored in the UML model. To automatically resolve the equations, Matlab was used as algebra system since most property values were matrices or vectors. The language attribute of the UML expression was set to "Matlab" so that only the Matlab-specific expressions were collected and evaluated. The results of the calculations were transferred back to the UML model. In the PPT instance of Fig. 8.18, the O_PPT_Thr vector was computed by Matlab.

A Simulink model was used for the simulation of the attitude and orbit control system (AOCS) of the satellite. The model simulated the behavior of environmental conditions, actuators, sensors and on-board computers. Different in-orbit satellite operations could thus be simulated. The impact of the thruster orientation on the AOCS of the satellite needed to be simulated accurately in advance. Since the access to the complete Simulink model of the Perseus satellite was limited due to proprietary toolboxes, the Simulink model was launched and accessed through a Matlab function.

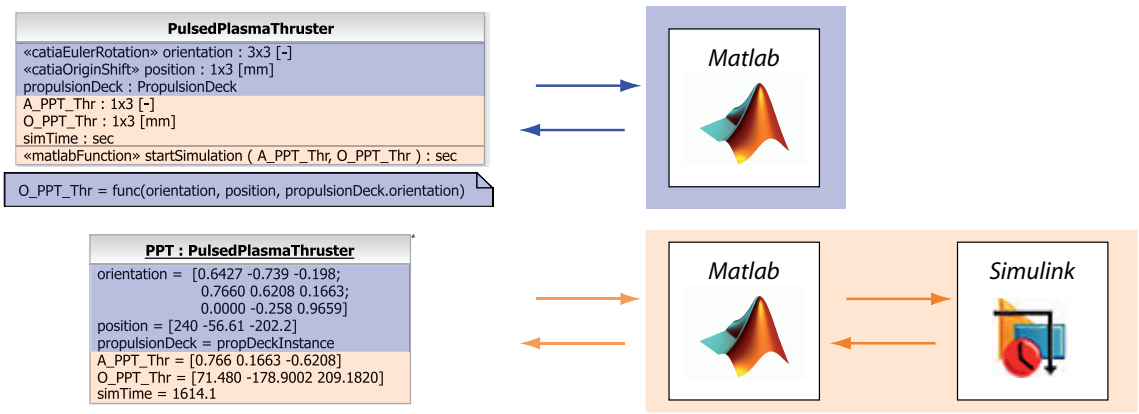

Figure 8.18: Interfaces between UML and Matlab/Simulink (after Gross et al. [54])

The Matlab function was represented in UML by an operation with the *«matlabFunction»* stereotype. In Fig. 8.18, the startSimulation operation can be seen in the PulsedPlasmaThruster class. The input and output arguments of the Matlab function were described as UML parameters. The Simulink simulation based on the O_PPT_Thr and A_PPT_Thr properties, describing respectively the thrust origin position and orientation, was launched through the Matlab startSimulation function. The related UML operation and properties are colored orange in Fig. 8.18. The time until the satellite reaction wheels, under the application of a specific thrust origin and direction, had saturated was the simulation result simTime which was written back into the UML model.

Changes in the CATIA-defined geometry of the satellite have an effect on its dynamic behavior described in Simulink. To automatically keep the Simulink model consistent with the CATIA model, a data exchange from CATIA to Simulink via the UML-based central product model had been implemented. After a change in the CATIA geometry, the new CATIA measures were imported back into the UML-based product model and the Simulink values were updated accordingly through the evaluation of UML constraints which linked CATIA-specific and Simulink-specific UML properties. The resulting Simulink simulation was then consistent with the CATIA-specific geometry of the satellite.

As displayed in Fig. 8.19, the Simulink-specific O_PPT_Thr property for example depended on the CATIA-specific orientation and position properties. The CATIA model of the pulsed plasma thruster (PPT) is shown in Fig. 8.19 left. The UML representation of the PPT is depicted (Fig. 8.19 middle) and the PPT-dependent Simulink blocks contributing to the dynamic behavior of the complete satellite are shown (Fig. 8.19 right). The UML PPT class contains two attributes tagged with CATIA-specific stereotypes which mark the import of the orientation and position measures of the CATIA PPT product.

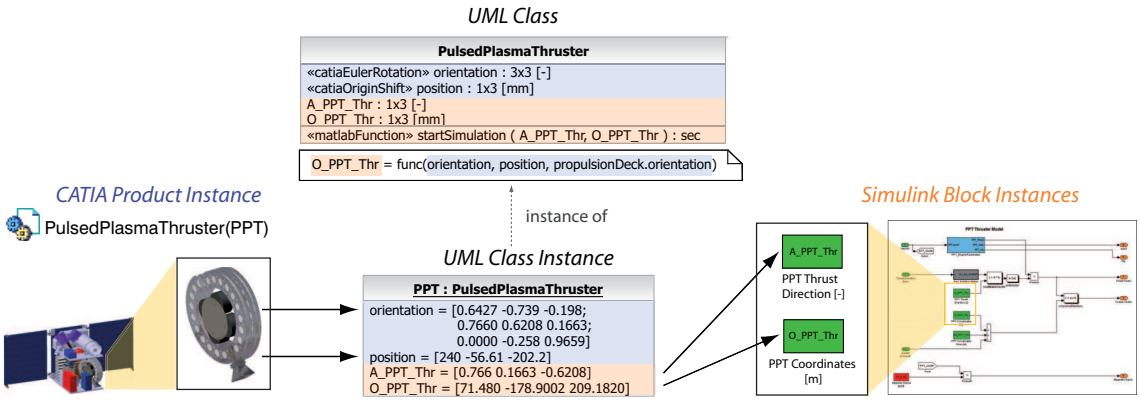

Figure 8.19: Consistency between CATIA- and Simulink-specific data through a common UML product model (after Gross et al. [54])

The UML instance contains the values imported from CATIA as indicated in blue. The values required for the initialization of the Simulink model are indicated in orange. The UML constraint, marked partially in orange and partially in blue, sets the link between the CATIA- and Simulink-specific properties.

The impact of different thruster orientations on the saturation time of the satellite reaction wheels was determined. This represented a test scenario including a dependency between a geometric configuration and a related dynamic system behavior. The orientation of the satellite is disturbed by the torque the thruster applies on the satellite. The reaction wheels of the attitude and orbit control system (AOCS) countervail the disturbance torque by increasing their rotation speed in order to keep the satellite aligned. Over time, the reaction wheels build up stored momentum that needs to be canceled. If the wheels have reached their maximum rotation speed, saturation is attained and the thruster has to be cut off. The aim is to maximize the saturation time of the reaction wheels by finding the optimal thruster orientation.

The process of sequentially simulating different satellite configurations was described through an executable activity diagram. The process consisted of evaluating different geometric satellite configurations according to their effect on the saturation time of the reaction wheels. Figure 8.20 shows the activity diagram for the automated evaluation of satellite configurations with different thruster orientations. The process steps were described in UML as call operation actions which were connected with UML object flows to determine their order of execution. Process input arguments corresponded to UML activity parameter nodes. In Fig. 8.20, process parameters are shown on the left hand side of the activity diagram. The upper input parameter contains the path to the UML model which is to be loaded. The two lower ones are used to describe the parameter name and value which have to be changed during the simulation of various satellite configurations.

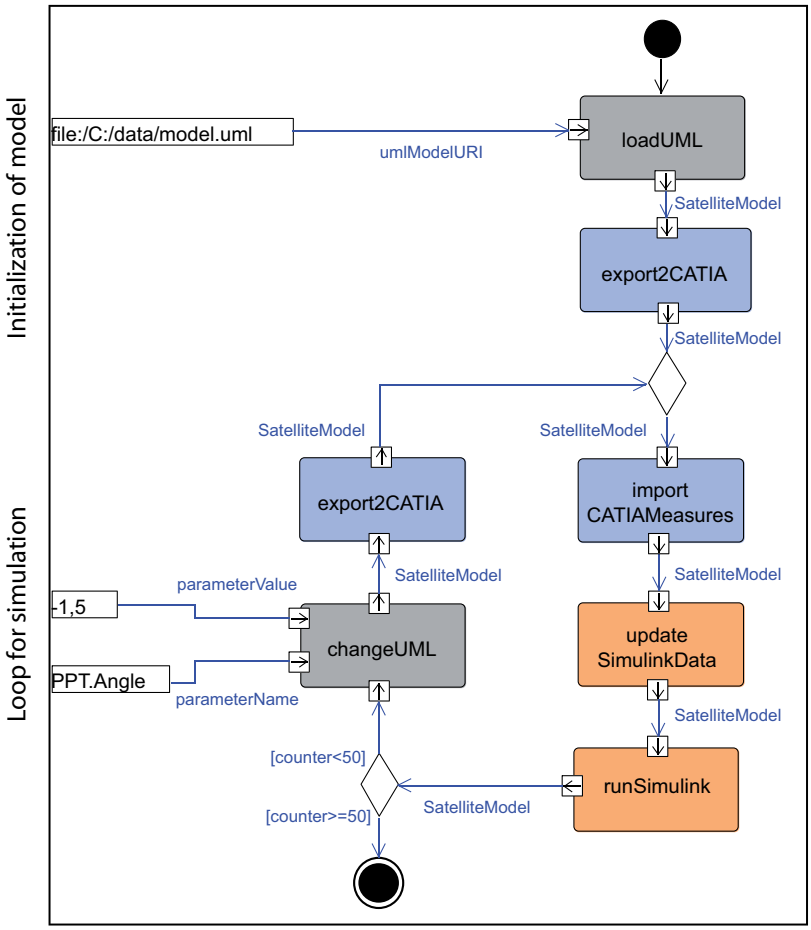

Figure 8.20: Activity diagram for the automated evaluation of different Perseus thruster configurations (after Gross et al. [54])

Thus every parameter of the UML-based central product model could be accessed and changed easily. In this example, the angle of the thruster was changed around the x-axis of the satellite in 1.5 deg steps (Fig. 8.21).

The activity diagram was executable by linking activity actions with Java methods which either modified the UML model or launched import/export interfaces. The execution of the UML activity diagram in Fig. 8.20 started at the black-filled UML initial node at the top of the diagram. The first action loaded the UML product model and the next one exported the first satellite configuration to CATIA. The thruster orientation had an impact on the thrust direction, the thrust origin, the center of gravity and the inertia of the satellite. These properties were therefore measured in CATIA and sent back into the UML model. UML constraints which linked CATIA-specific and Simulink-specific properties were then evaluated by Matlab. The Simulink simulation which determined the saturation time was then launched based on updated CATIA-specific measures. The simulation results were stored back in the UML model. The decision node, depicted as a rhombus

in the lower left corner, either ended the evaluation loop or started a new evaluation of a new satellite configuration with a new parameter value. The CATIA- and Simulink-based evaluation of a satellite configuration via the UML-based central product model was then repeated.

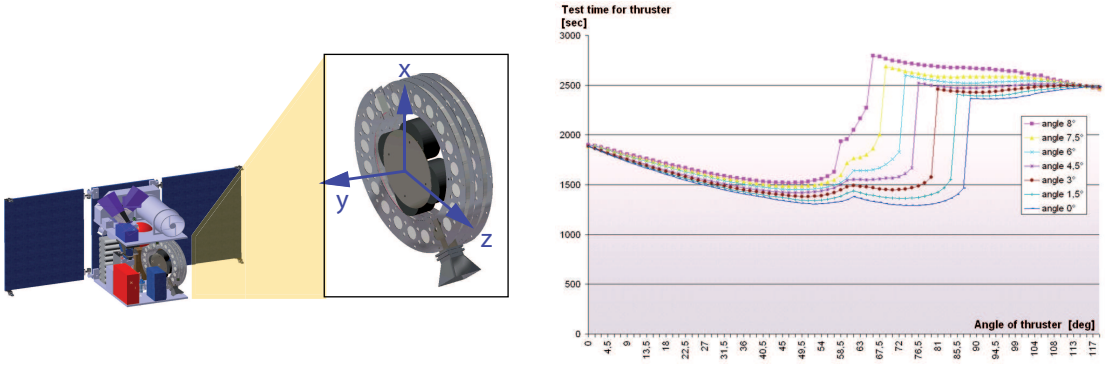

Figure 8.21: Results of the evaluation of different geometric configurations (after Gross et al. [54])

Figure 8.21 shows the results of the automated evaluation of different configurations. Each point in the graph represents a different geometric thruster configuration. On the horizontal axis, the orientation angle of the thruster around the y-axis is drawn. On the vertical axis, the resulting saturation time of the reaction wheels is depicted. The different colors stand for a change of the thruster angle around the x-axis of the satellite. The saturation time depends on the lever arm of the disturbance torque created by the thrust and on the satellite position within the earth magnetic field.

This Section presented the UML lightweight extensions necessary to describe geometric features authored in CATIA and control system features authored in Matlab/Simulink in a common UML model. Update mechanisms were implemented to support a quick reconfiguration of models and avoid a time-consuming generation of complete models from scratch. Furthermore, the UML-based product model was also used in the context of automated design. The evaluation of a series of different thruster orientations according to the saturation time of the reaction wheels was described as an executable UML activity diagram. The sequential import/export procedures between the application-specific models and the central UML-based product model were thereby executed automatically.

8.4 Generation of aircraft geometries

The generation of consistent geometric models of different aircraft configurations is typically a time-demanding effort. This Section presents the UML-based generation of customizable aircraft geometries [18, 19]. The project was undertaken in partnership with the Institute of Aerodynamics and Gasdynamics[4] of the University of Stuttgart. The geometric models can be used subsequently for an aerodynamic analysis of different geometric aircraft configurations.

The definition of an aircraft geometry was based upon points and a limited set of geometric operations. Points were chosen as they represent the smallest possible geometric entity and thus allow to define a large variety of different aircraft geometries. A limited set of geometric operations was used to create volumes based upon points. The operations are shown in Figure 8.22. Sections were defined based on a series of points. Similarly, guiding lines were defined based on points and gradient conditions. Consequently, volumes were defined based on a selection of sections and guiding lines. This sequence of geometric operations was used to generate volumes corresponding to various aircraft parts. Figure 8.22 presents the definition of fuselage and wing parts based on points.

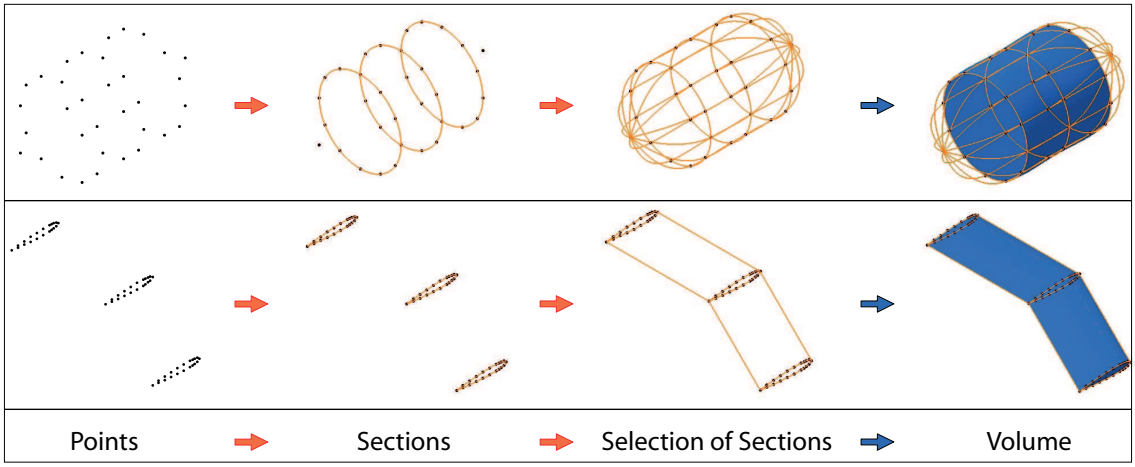

Figure 8.22: Definition of volumes based on points and sections (after Boehnke et al. [19])

The aircraft geometry was defined in CATIA. The points represented the main building blocks of the aircraft geometry and were positioned according to cartesian coordinates in CATIA parts through CATIA user defined features. The geometric operations to create volumes were invoked by the execution of CATIA scripts. Only the engine nacelle was predefined as a CATIA user defined feature. All other aircraft parts were dynamically generated based on points and scripts. The CATIA-specific geometric model of the air-

[4]Institut für Aerodynamik und Gasdynamik (IAG), `http://www.iag.uni-stuttgart.de/`

craft was generated from scratch based upon a customizable UML product model. The next paragraphs present the main geometric elements of the aircraft geometry and their counterparts in UML.

The points were categorized according to the type of section they represented. Figure 8.23 for example represents the PointOnCircle and PointOnEllipse point classes. Both inherited from the common Point class with the Xccord, Ycoord and Zcoord properties as all points were defined by cartesian coordinates. The sections could include a variable number of points. The section type-specific classes therefore included symbolic equations in the form of UML constraints in order to compute the point coordinates as a function of the variable number of section points. Figure 8.23 examplarily shows the computation of the y and z coordinates respectively for a point on a circle and a point on an elliptical section. Besides being computed by symbolic equations, the point coordinates could also be imported from Excel spreadsheets. Figure 8.24 presents all the different section type-specific point classes which allow to compute the point coordinates for a specific section type according to specific symbolic equations. Points on wing-specific sections thereby belonged to the abstract PointOnProfile class and more specifically to a concrete point location-specific class such as PointOnProfilefromNACAbacklow or PointOnProfilefromNACAfrontup.

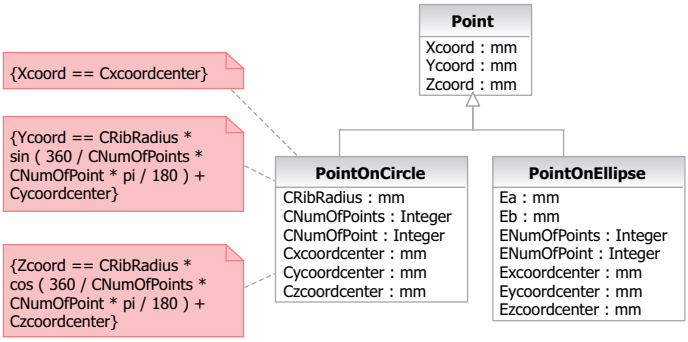

Figure 8.23: Symbolic equations represented as UML constraints (after Boehnke et al. [19])

The sections were categorized in profiles and ribs which were represented respectively in UML through a Profile and a Rib class. The Profile class was further specialized according to wing type-specific profiles, such as WingProfile or FlapProfile. The types of volumes, which were created based on the sections, were classified according to the aircraft parts they described, such as wings, fuselages, flaps and intersection elements. Their geometric representation is depicted in Fig. 8.25 and their corresponding abstract UML representation as UML classes is shown in Fig. 8.24.

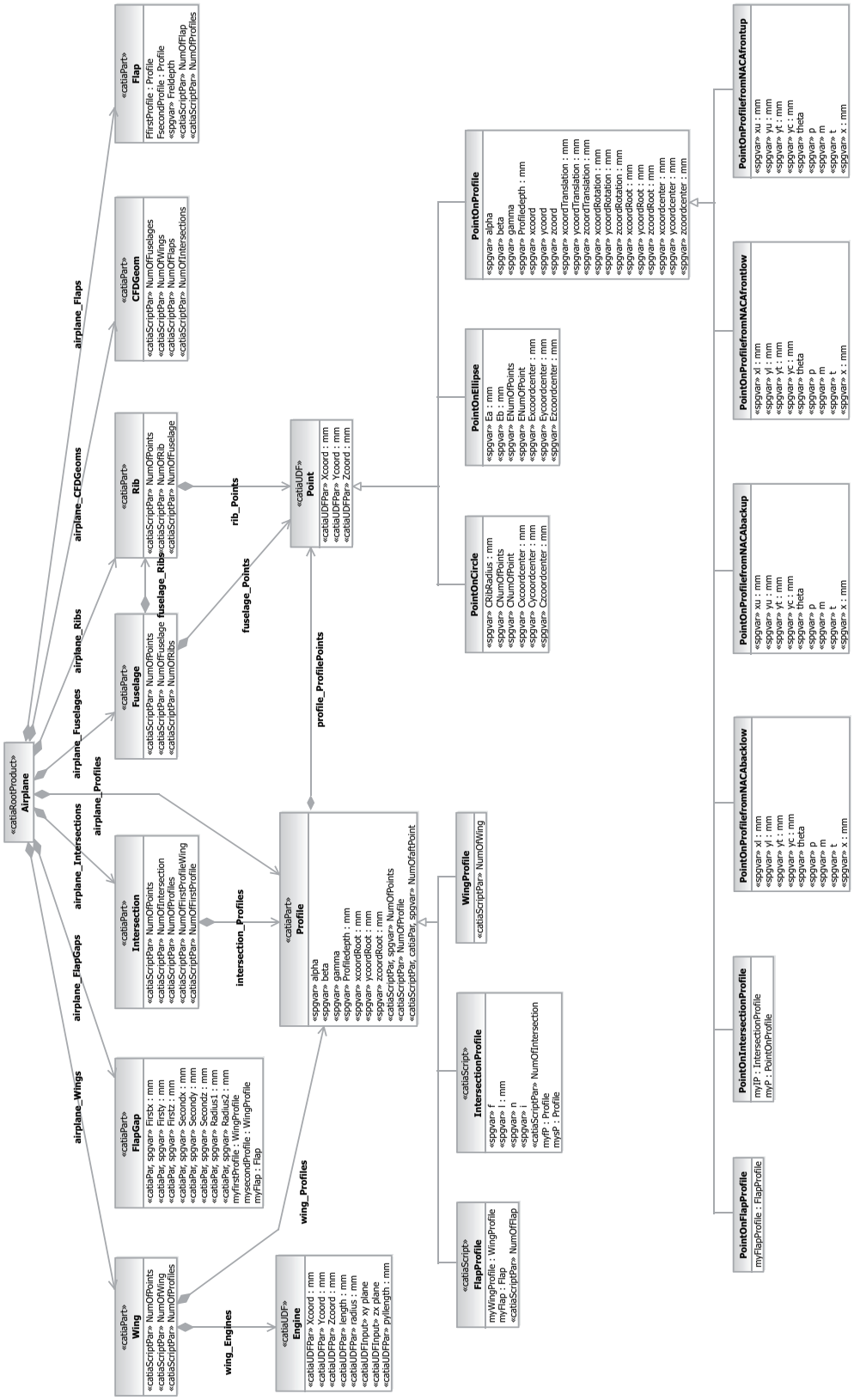

Figure 8.24: UML class diagram of geometric aircraft entities (after Boehnke et al. [19])

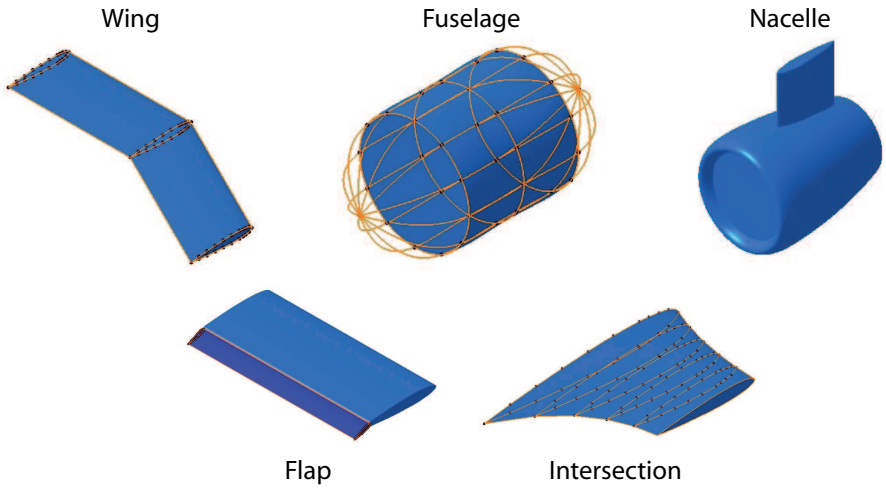

Figure 8.25: Main geometric parts of the aircraft geometry (after Boehnke et al. [19])

The fuselage part consisted of rib sections as well as front and aft points. The rib sections were aligned along the longitudinal axis of the fuselage and were of circular or elliptical shape. The front and aft points allowed to define guiding lines with gradient constraints in order to design tapered or smooth fuselage ends. The guiding lines were splines which followed the longitudinal axis of the fuselage from the front point to the aft point by passing through all rib points having the same index. CATIA scripts then formed the volume corresponding to the fuselage part based on the rib sections and the longitudinal guiding lines.

The wing part consisted of profile sections which were parameterized in order to describe different airfoils. The profiles were customized according to their depth, thickness, maximum camber and camber position. Furthermore, their orientation was specified by Euler angles. The profiles were defined by splines which connected all profile points. CATIA scripts generated volumes based on profiles and on polylines connecting the profiles. The resulting volumes represented wing parts. The engine nacelle part was the only part which was predefined as a parameterized user defined feature. The nacelle could be adapted according to its radius and length as well as its position relative to the wing. The nacelle could thus be placed under or over a wing.

Intersection parts were defined identically to wing parts. They were placed for example between the wings and the fuselage in order to define a realistic geometry. Boolean operations were then used to remove the volume of the intersection element which coincided with the neighboring parts. Flap parts were very similar to wing parts. However, their geometric representation was more constrained as they only consisted of two profiles. The flaps were usually embedded in the wing parts. A flap-gap element in the form of a cylinder was introduced in the geometric model of the aircraft. The volume of the

wing part which coincided with the flag-gap element was then removed through a boolean operation to leave space for the placement of the flap.

The geometry of the aircraft was described as a CATIA product which included CATIA part instances representing the various aircraft parts. The CATIA part instances were generated dynamically through CATIA user defined features and CATIA scripts. As described in Chapter 4, CATIA products, parts, user defined features and scripts were represented in UML through UML classes with appropriate stereotypes. The instances of CATIA features were represented accordingly in UML through UML instances. The UML-based product model could thus represent CATIA-specific features which could be automatically translated into a CATIA-specific model of the aircraft geometry. The UML-based product model was easier to customize than the detailed application-specific CATIA model as it only represented geometric information which was subject to change.

Excel cell values were imported into the UML model. The symbolic equations within the UML-based product model were evaluated by a computer algebra system in order to compute the property values of UML instances, such as point coordinates. The UML-based product model could then be exported to CATIA.

The geometry of an aircraft requires the definition of many geometric elements such as a multitude of sections and points. Manual modifications within a UML or corresponding CATIA model were thus too time-consuming. A Java program was therefore used to generate a UML model of an aircraft geometry which could be easily modified in order to insert, remove, or displace points and sections. The Java program referred to Java methods for common tasks such as the insertion of UML instances representing points, ribs, profiles and larger volumes like wings. Lines of code to create the geometry of an Airbus A321 are for example shown in Fig. 8.26 left. The Java program for example referred to the createRibInstanceFromCircle Java method whose Javadoc description is represented in Fig. 8.26 right.

Main Java Program

```
...
//Main Fuselage
AirplaneRules.createRibInstancefromCircle(airplaneModel,1   ,1, 10    ,0       ,0   ,-200  ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,2   ,1, 900   ,900     ,0   ,-200  ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,3   ,1, 1100  ,1270    ,0   ,-200  ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,4   ,1, 1975  ,4699    ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,5   ,1, 1975  ,4700    ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,6   ,1, 1975  ,4701    ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,7   ,1, 1975  ,10000   ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,8   ,1, 1975  ,15000   ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,9   ,1, 1975  ,20000   ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,10  ,1, 1975  ,25399   ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,11  ,1, 1975  ,25400   ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,12  ,1, 1975  ,25401   ,0   ,0     ,12);
AirplaneRules.createRibInstancefromCircle(airplaneModel,13  ,1, 200   ,37000   ,0   ,1775  ,12);
...
```

Javadoc of createRibInstancefromCircle(...)

Create Rib Instance with PointOnCircle Instances

Parameters:
airplaneModel AirplaneModel to be returned
NumOfRib Number of the Rib
NumOfFuselage Number of the Fuselage the Rib is in
radius Radius of the Circle
xcenter X Center Position of the Circle
ycenter Y Center Position of the Circle
zcenter Z Center Position of the Circle
PointIndex Number of Points to be created
Returns:
airplaneModel

Figure 8.26: Java code for the creation of UML instances referring to fuselage ribs

Figure 8.27 presents a sample of various aircraft geometries that were generated based upon a highly customizable UML model of an aircraft geometry. The aircraft models represent to a large extent, starting clockwise from the bottom left, Boeing's X-48B Blended Wing Body demonstrator, the Bell X-1, a conventional aircraft, the family of Airbus A318/319/320/321 and a fictional artistic scramjet model. The latter example was for example conceived and implemented, in collaboration with a member of the Scramjet Research Group at the University of Stuttgart[5], only within a few hours. The variety of generated examples showed the customizability of the UML-based central product model.

Figure 8.27: Various aircraft geometries based on a UML product model (after Boehnke et al. [19])

This Section has shown the representation of low-level geometric points and operations in a UML-based product model. The UML model included more than geometric information by capturing Excel-specific point coordinates as well as by computing the point coordinates of predefined section types based on symbolic equations. The UML-based product model was programmatically edited by a Java program as the UML model of an aircraft geometry is too large to be created manually in a reliable and efficient manner. Changes within the Java program led to modifications in the UML model and consequently to the resulting CATIA-specific geometric aircraft model. The variety of generated examples showed the customizability of the UML-based central product model.

[5]Graduiertenkolleg-SCRamjet, `http://www.uni-stuttgart.de/itlr/graduierten/`

8.5 Summary

This Chapter presented the use of a UML-based product model in design scenarios which included multiple inter-model dependencies and model modifications. The test cases showed the representation of diverse geometric information within the UML-based product model. The geometric information within the use case scenarios consisted of coarse-grained geometric data, such as the simple rectangular and cylindrical boxes for the representation of cabin pressure system units, as well as fined-grained geometric data, such as the highly detailed geometric primitives and operations to describe aircraft parts. Furthermore, the design of conveyor systems and of the Perseus satellite included geometric data from a professional industrial context. Besides geometry, the use cases presented the integration of Excel-specific spreadsheet data, as well as Matlab/Simulink-specific dynamic system models and symbolic equations to achieve data consistency within the UML-based product model. The UML-based product model supported the integration of diverse product information and proved its adaptability to describe various product configurations. Furthermore, the test cases demonstrated the capability of translating UML model information into application-specific models.

In all test cases, modifications to the UML-based product model were defined formally in an executable format. The non-proprietary Java API of UML models was used to describe recurrent design steps or design decisions in Java methods. Furthermore, the test cases related to the design of conveyor systems and the Perseus satelite represented design processes through UML activity diagrams. The diagrams enabled a better overview of the design requirements and design steps as well as of the various information flows within a design process. The executability of UML activity diagrams was supported by linking activity actions with predefined Java methods. The execution of UML activity diagrams resulted in the generation of UML-based product models and subsequent application-specific models which were conform to the original product requirements.

Chapter 9

Conclusion

Section 9.1 reviews the motivation to investigate a UML-based central product model for the management of multidisciplinary dependencies during product design. Section 9.2 summarizes the results regarding the mapping of discipline- and application-specific models into a UML-based central product model and the use of the UML-based central product model for the design of customized multidisciplinary products. Finally, Section 9.3 presents an outlook on the future harmonization of modeling languages as well as the potential extensions of the UML-based central product model.

9.1 UML-based central product model

Computer-aided design of multidisciplinary products involves the use of specialized discipline-specific software applications in order to model and simulate various product aspects. Dependencies between models are thereby frequent as the same product information often appears redundantly in various engineering models. A change in one model then requires the update of dependent models. Otherwise, the simulation of models based on inconsistent data will lead to meaningless or misleading results and subsequent wrong design decisions. Synchronization of models may involve large quantities of data. The manual update of models by engineers therefore represents an error-prone and time-consuming task. Hence, a framework for automatic model updates is needed to guarantee data consistency across all product models.

Data consistency between models is achieved automatically through model-to-model data exchange software. However, the development and maintenance of each specific data exchange connection represents a large effort. The use of a central product model enables a reduction in the required number of data exchange connections by acting as the hub in a hub-and-spoke network. The central product model stores redundant product informa-

tion which is spread across several models and maintains data consistency through data exchange connections between itself and each specific model. For example, in a scenario with *n* specific models, the bidirectional linking of models via a central product model requires only *2n* connections while *n(n-1)* connections are necessary for the equivalent direct linking of models .

Most current central product models are dedicated to the design of specific products. However, some central product models are generic enough to be employed for the design of different product types. This is advantageous since the same central product model and associated application-specific translation software can be reused for the design of various products across a wide range of industry sectors. Among the generic central product models none has yet gained wide acceptance nor reached the status of an international standard. Thus no standard for example currently exists for the integration of mechanical, electronic and software information in a central product model that could support the design of mechatronic products. As software and electronics are more and more embedded in conventional mechanical products, the development of a standard central product model is of utmost importance for the design of mechatronic products ranging from aerospace engineering to modern manufacturing facilities.

In general, the acceptance of a model and its modeling concepts depends on the simplicity with which specific aspects of a system can be described. In the case of the central product model, modelers need to easily describe discipline-specific information and inter-model dependencies. Most engineers are specialized in specific disciplines and are only familiar with specific models and related modeling concepts. However, the central product model cannot represent diverse product aspects identically as in specialized discipline- and application-specific models because it cannot support the multitude of various discipline- and application-specific modeling concepts. Central product models are therefore comprised of a limited manageable set of modeling concepts that correspond to generalizations of specific modeling concepts. The choice of generalized discipline- and application-independent modeling concepts is critical for the capability of the central product model to represent product models from a wide range of disciplines and modeling applications.

Although models of different engineering disciplines are highly diverse, most models which are edited with current state-of-the-art software applications share common modeling concepts in order to support modular design. The capacity to easily exchange and reuse model components across several models promotes flexibility and productivity in modeling. This avoids the time-consuming creation of models from scratch. Most models therefore share common modeling principles in order to describe modular model compo-

nents. Common characteristics of modules include in- and outputs, hidden and visible outward-facing information, as well as templates and instances.

There is currently no widely accepted standard to represent with general overarching concepts modules from different disciplines. However, modularity is especially important in software engineering. Software design is an engineering discipline in which changes and updates are more frequent than in other engineering disciplines as software code is simpler to modify than tangible mechanical engineering components which require manufacturing and resources. As a result, concepts that promote modular design are more commonly found in software engineering than in other engineering disciplines. Sophisticated programming concepts have been developed in software engineering to support modularity. The most prominent are the object-oriented programming concepts which consist of encapsulating variables and functions into modular units called objects. Graphical models of object-oriented software represent the classification, communication and internal structure of software objects.

Although object-oriented modeling concepts are currently mainly used for software modeling, they are generic and can be used in the context of a central product model to describe the common modular structure of models from different engineering disciplines. Engineers can then recognize their discipline-specific information within the larger central product model due to the common modular structure of the specific model and the central product model.

As a central product model is to be used across several disciplines, it addresses many parties and requires standardization because special training and dedicated conversion tools are needed. Besides, a standardized central product model would be desirable as it would not only eliminate the confusion caused by different central product models but also reduce the development costs of translation software.

Since its emergence in 1997, the Unified Modeling Language (UML) has been the de facto standard for object-oriented modeling and is widely used in software engineering. This thesis investigated the capability of the UML to describe the common modular structure of various discipline-specific product models within a central UML-based product model. The integration of geometric, controller and multibody system models is required in many mechatronic products which abound in aerospace, automobile or manufacturing products. The approach of reusing the UML to support the representation of various models in a central product model was examined by representing state-of-the-art application-specific geometric, controller and multibody system models in a common UML-based product model.

9.2 Results

Geometric, controller and multibody system models are edited according to specialized software applications. In order to recognize and interpret the application-specific models within the central UML-based product model, lightweight UML extensions in the form of stereotypes were applied on top of the generic UML modeling entities in order to denote application-specific information. Stereotypes which corresponded to a specific application were regrouped in a UML profile. This thesis defined application-specific UML profiles to represent in a common UML-based product model application-specific geometric, controller and multibody system models. The approach was demonstrated for widely used state-of-the-art modeling applications. The mapping rules between the application-specific models and the central UML model were implemented and tested in various design projects in order to support consistency across various product models as well as to contribute to the efficient design of customizable products.

The UML profiles for geometric models captured commonly shared geometric product information, such as volume, mass, center of gravity and moment of inertia. This type of information is typically stored redundantly in various models such as in geometric and multibody system models or spreadsheets. The representation of the redundant geometric information centrally in the UML model makes it possible to keep it consistent across various specialized product models. A change in a specific model can for example be forwarded to the central model which can then update other specific models which require synchronization. In addition, application-specific modeling concepts were represented in the UML model in order to support the automatic translation of the UML-based geometric information into application-specific geometric models. Application-specific modeling features such as parts, assemblies, assembly constraints, part dependencies, features and geometric primitives were therefore mapped into the UML-based product model.

High-level object-oriented geometric modeling concepts were described in UML by their homologous generalized UML modeling concepts with a corresponding applied stereotype while detailed low-level geometric entities were represented in UML as instances of predefined geometric types. This mapping logic allowed to represent geometric modeling concepts in UML and vice versa according to a one-to-one correspondence. Models from widespread 3D geometry modelers such as CATIA and SolidWorks as well as the open VRML format were mapped into UML. As most CAD modeling applications share a large common set of modeling concepts, the presented mapping rules could also be extended to other 3D geometric modeling applications.

The test cases in Chapter 8 included a variety of geometric models: models with highly detailed geometric entities, such as for the design of a variety of aircraft, as well as

models originating from existing industrial design projects, such as the geometric models related to the design of the PERSEUS satellite and conveyor system configurations. Furthermore, the slider-crank mechanism test case of Chapter 4 included various types of dependencies between geometric parts. All test cases showed the representation of geometric model information in UML and the subsequent automatic UML-based generation of application-specific geometric models. In addition, the test cases related to the design of the slider-crank mechanism and the Perseus satellite showed the automatic import of geometric model information into the UML model.

Next to geometric models, dynamic system and multibody system models were mapped into UML. Both Simulink-specific dynamic system models and SimMechanics-specific multibody system models are block diagrams composed of block types, block instances and edges between block instances. However, the edges which connect the blocks represent in the Simulink dynamic system model signals, in other words information flows, while they represent static connections in the SimMechanics multibody system model. Simulink-specific dynamic system models were therefore mapped into UML activity diagrams and SimMechanics-specific multibody system models into composite structure diagrams.

The block types of both Simulink and SimMechanics models represent templates. They were therefore both mapped into UML classes. However, Simulink blocks represent a behavior in contrast to SimMechanics blocks which describe static bodies and joints. Simulink block types were therefore mapped into UML activities which are specialized UML classes. Simulink-specific block instances were mapped into UML actions and SimMechanics-specific block instances into UML parts. In addition, block instances were also mapped into UML instances. Furthermore, the Simulink-specific edges between block instances were represented as UML object flows between action pins while the SimMechanics-specific edges were depicted as UML connectors between parts. Apart from block instances, the mapping of Simulink and SimMechanics modeling concepts into corresponding UML modeling concepts was bijective, in other words according to a one-to-one correspondence. The dynamic and multibody system models shared great resemblance with their corresponding UML diagrams and therefore allowed a mostly intuitive mapping.

The mapping of Simulink, SimMechanics and combined Simulink/SimMechanics models into UML was applied to the slider-crank mechanism example of Chapter 5. As the blocks within the Simulink and SimMechanics models contain by default numerous detailed information, the import of existing models into UML was more efficient than describing in UML the Simulink- and SimMechanics-specific information from scratch.

The translation of UML models into Simulink- and SimMechanics-specific models was validated by generating models identical to the imported ones, in other words by performing a round-trip transformation.

UML stereotypes were defined in order to refer to external data in Excel spreadsheets. The test cases related to the evaluation of cabin pressure systems in Section 8.1 and the generation of aircraft geometries in Section 8.4 for example required the import of Excel data. Similar UML stereotypes could be specified in order to refer to values from other data sources such as databases. Furthermore, a UML stereotype was defined to refer to external Matlab-specific functions. This allowed to perform complex computations and return the results back to the UML-based product model. The automated evaluation of satellite configurations in Section 8.3 for example required the invocation of Matlab functions for matrix manipulations. Similarly, references to external functions other than Matlab could be represented in UML through UML operations with appropriate stereotypes.

This thesis demonstrated that the UML can be used beyond conventional software modeling to establish a standard central product model. The object-oriented UML modeling concepts corresponded semantically to the modeling concepts of various disciplines including geometric, dynamic and multibody system models. This facilitated the representation of discipline- and application-specific model information in a central UML-based product model.

Inter-model dependencies were described in the common UML-based product model. Inter-model dependencies between features of various discipline- and application-specific product models were represented as dependencies between related UML properties. Dependencies between UML properties were described through UML constraints which referred to algebraic equations. Their resolution was achieved through a solution path generator algorithm and a computer algebra system. The resolution of the UML constraints established data consistency within the central UML-based product model and was used in all test cases.

The automated creation or modification of the UML-based central product model was enabled by an Application Programming Interface (API) in Java and was used in all presented test cases. Furthermore, UML activity diagrams were used to graphically describe design processes. The graphical representation of a design process through a UML activity diagram is easier to understand than an equivalent textual representation through a Java program. A design process described as a UML activity diagram can thus be understood by more parties. In order to achieve the same level of executability as with Java programs, nodes within UML activity diagrams referred to Java methods. An executable

UML activity diagram was used for the automated design of conveyor system configurations (Sec. 8.2) and the automated evaluation of satellite configurations (Sec. 8.3).

The easily extensible open-source Eclipse platform was used to integrate UML translators and editors in a single software environment. The implementation of the various translators between the UML model and the application-specific models as well as the software for the executability of UML activity diagrams were packaged as Eclipse plugins which extended the Eclipse platform. Furthermore, various open-source UML editors, which were available as Eclipse plugins, were also used to extend the Eclipse platform. As a result, a single software environment based on the Eclipse platform regrouped the features to perform UML-based product design.

The UML-based central product model represented application-specific modeling concepts in order to enable an automatic translation of the UML representation of application-specific model information into detailed discipline- and application-specific models for simulation. As a result, changes in the UML-based product model were automatically propagated to detailed application-specific models. Consistency between the central UML-based product model and the application-specific models was guaranteed. This enabled to efficiently generate a multitude of different consistent model configurations which were used to evaluate various product configurations. The customizable UML-based central product model was used in all presented test cases. As an example, the customization of the UML-based central product model of the slider-crank mechanism in Section 7.2 allowed to automatically generate various consistent geometric and multibody system models which involved the update of many values. The benefits of the centrally defined UML model customization were especially visible in the project related to the generation of various aircraft geometric models in Section 8.4. A variety of aircraft geometric models, which differed in size and topology, were generated from a customizable UML-based product model. They represented a wide range of aircraft including a fictional artistic scramjet, Boeing's X-48B Blended Wing Body demonstrator, the Bell X-1 and the family of Airbus A318/319/320/321.

9.3 Outlook

By integrating product information, the central product model represents a central data repository which facilitates the use of knowledge-based engineering or multidisciplinary optimization frameworks since they only need to address the central product model instead of many separate engineering models. Through a tight linkage between the UML-based product model and state-of-the-art engineering software applications, multidisci-

plinary design processes requiring several iterations to reach an optimal product configuration could be fully automated. The UML-based product model could therefore be easily combined with a rule-based automated design compiler approach as in Alber and Rudolph [2, 141].

The choice of the UML as a product modeling language enables to use the Model Driven Architecture (MDA) concepts [112] - originally intended for software design - for a formal engineering design process. The MDA is a framework for using modeling standards such as the UML in software development. It provides an approach to reuse abstract software models for the automatic generation of various platform-specific software models or code through model transformations. Engineering decisions or design rules could similarly be described as model transformations and be executed by current tools supporting MDA. Kerzhner and Paredis [83] for example applied graph-based model transformations on MDA-compliant models to generate different alternatives of hydraulic circuits.

The development of further domain-specific or application-specific UML extensions covering different product lifecycle aspects such as requirements, manufacturing and costs would increase the integrative role and value of the UML-based product model. In addition, the presented application-specific UML extensions could be further improved to include more modeling concepts.

A future standardization of application-specific UML profiles and their respective mappings is required in order to share these on a large scale with many parties. Application-specific UML extensions should ideally be defined in cooperation with application developers and the Object Management Group (OMG) which is a consortium aiming at setting standards such as the UML. Furthermore, the mapping of application-specific models into UML could be described through the Query/View/Transformation (QVT) standard [119] of the OMG.

Similarities between UML, OWL and other modeling languages such as EXPRESS exist. It is therefore probable that the different modeling languages which were developed over the years in distinct disciplines, such as UML in software design, EXPRESS in data modeling and OWL in artificial intelligence, will probably undergo a harmonization process in the near future. Approaches are undertaken to facilitate interoperability between UML, EXPRESS and OWL. Very widespread state-of-the-art UML modeling tools could then be used for EXPRESS as well as for OWL modeling. Approaches for a narrower coexistence and integration of EXPRESS and UML are therefore investigated [92]. The Mexico[1] project for example is developing a new EXPRESS metamodel which would,

[1]MOF 2 Based EXPRESS Integration and Coexistence, http://www.modelalchemy.com/

just as the UML metamodel, be an instance of the MOF metamodel. Similarly, Kiko and Atkinson [84] have compared in detail UML and OWL to facilitate their harmonization. Atkinson [7] is in favor of developing a core level unification of UML and OWL as there is no fundamental difference between modeling and ontology representation. Ontologies have been defined in UML in order to use common UML tools for ontology design [30, 90, 48]. UML models have thereby been transformed into executable logic representations [42].

As described in Section 3.5, an important new extension of UML is SysML [118] for systems engineering. SysML includes additional modeling constructs to describe system requirements, behavior, structure and parametrics. But SysML is a new modeling language which has not yet reached a mature status similar to UML. Changes in the new SysML modeling language are therefore highly probable in the near future. Eventually, SysML will most probably be better suited than UML to establish a standard central product model. As SysML is based on UML, both languages share many common modeling concepts. Current UML profiles could therefore be reused to a large extent in SysML models.

Appendix A

Tables of correspondence between modeling concepts

CATIA-specific modeling concept	UML modeling concept	Stereotype	Section
Part	Class	«$catiaPart$»	4.1.1
Part instance	Instance	-	4.1.1
Part Parameter	Property	«$catiaPar$»	4.1.2
Part Measure	Property	e.g. «$catiaMass$»	4.1.2
Publication	Interface	«$catiaPublication$»	4.1.3
CCP Link	Usage	«$catiaCCPLink$»	4.1.3
Import Link	Usage	«$catiaImportLink$»	4.1.3
Product	Class	«$catiaProduct$»	4.1.4
Product instance	Instance	-	4.1.3
Assembly Constraint	Constraint	e.g. «$catiaCoincidence$»	4.1.5
PowerCopy	Class	«$catiaPowerCopy$»	4.1.6
User Defined Feature	Class	«$catiaUDF$»	4.1.6
Script	Class	«$catiaScript$»	4.1.7

Table A.1: Table of correspondence between CATIA and UML modeling concepts

SolidWorks-specific modeling concept	UML modeling concept	Stereotype	Section
Part	Class	«sldWorksPart»	4.2.1
Assembly	Class	«sldWorksAsm»	4.2.1
Geometric entity type	Class (predefined in profile)	-	4.2.2
Geometric entity instance	Instance	-	4.2.2
Mate	Constraint	«sldWorksMate»	4.2.3

Table A.2: Table of correspondence between SolidWorks and UML modeling concepts

VRML-specific modeling concept	UML modeling concept	Stereotype	Section
Node type	Class	-	4.3.2
Node instance	Instance	-	4.3.2
VRML Transform Node	Class	«vrmlTransformNode»	4.3.3
VRML Transform Node Property	Property	e.g. «vrmlRotation»	4.3.3

Table A.3: Table of correspondence between VRML and UML modeling concepts

Simulink-specific modeling concept	UML modeling concept	Stereotype	Section
Model	Activity	«simulinkModel»	5.1.1
Block type	Activity (predefined in profile)	-	5.1.2
Block instance	Action + Instance	e.g. «integrator»	5.1.2
Block port	Pin	«simulinkPort»	5.1.2
Signal	ObjectFlow	«simulinkSignal»	5.1.3

Table A.4: Table of correspondence between Simulink and UML modeling concepts

SimMechanics-specific modeling concept	UML modeling concept	Stereotype	Section
Model	Class	«simMechModel»	5.2.1
Block type	Class (predefined in profile)	-	5.2.2
Block instance	Part + Instance	e.g. «ground»	5.2.2
Block port	Port	e.g. «simMechCS»	5.2.2
Connection	Connector	«simMechConnection»	5.2.3

Table A.5: Table of correspondence between SimMechanics and UML modeling concepts

Excel-specific modeling concept	UML modeling concept	Stereotype	Section
Cell value	Property or Slot	«excelCellValue»	6.1

Table A.6: Table of correspondence between Excel and UML modeling concepts

Matlab-specific modeling concept	UML modeling concept	Stereotype	Section
Function	Operation	«matlabFunction»	6.2
Function argument	Parameter	-	6.2

Table A.7: Table of correspondence between Matlab and UML modeling concepts

Equation-specific concept	UML modeling concept	Stereotype	Section
Equation	OpaqueExpression	«matlabFunction»	6.3
Equation variable	Property	«spgvar»	6.3
Equation fixed variable	Property	«spgconst»	6.3

Table A.8: Table of correspondence between equation and UML modeling concepts

Bibliography

[1] ABRAMOVICI, M. Future trends in product lifecycle management. In *The Future of Product Development. Proceedings of CIRP'07, Berlin* (2007).

[2] ALBER, R., AND RUDOLPH, S. "43" - a generic approach for engineering design grammars. In *Proc. of the AAAI Spring Symposium - Computational Synthesis, Stanford* (2003).

[3] ANDERL, R., JOHN, H., AND PÜTTER, C. EXPRESS. *Handbook on Architectures of Information Systems. Springer* (1998), 59–80.

[4] ANDERL, R., AND TRIPPENER, D. *STEP, STandard for the Exchange of Product Model Data*. Teubner Verlag, 2000.

[5] ARÉVALO, G., FALLERI, J.-R., AND NEBUT, M. H. C. Building Abstractions in Class Models: Formal Concept Analysis in a Model-Driven Approach. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), LNCS, Springer* (2006).

[6] ASSOCIATION FRANCAISE DE NORMALISATION (AFNOR). NF Z68-300 Automatisation industrielle - Spécifications du standard d'échange et de transfert (SET), 1993.

[7] ATKINSON, C. Unifying MDA and Knowledge Representation Technologies. In *The Model-Driven Semantic Web Workshop (MDSW 2004), September, Monterey CA* (2004).

[8] ATKINSON, C., AND KÜHNE, T. Model-Driven Development: A Metamodeling Foundation. *IEEE Software 20(5)* (2003), 36–41.

[9] BAADER, F., CALVANESE, D., MCGUINNESS, D. L., NARDI, D., AND PATEL-SCHNEIDER, P. F., Eds. *The description logic handbook : theory, implementation, and applications*. Cambridge University Press, 2007.

[10] BALDWIN, C. Y., AND CLARK, K. B. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

[11] BALMELLI, L. An Overview of the Systems Modeling Language for Products and Systems Development. *Journal of Object Technology 6(6)* (2007), 149–177.

[12] BATRES, R., WEST, M., LEAL, D., PRICE, D., MASAKI, K., SHIMADA, Y., FUCHINO, T., AND NAKA, Y. An upper ontology based on ISO 15926 . *Computers & Chemical Engineering 31(5-6)* (2007), 519–534.

[13] BAZJANAC, V. Building energy performance simulation as part of interoperable software environments . *Building and Environment 39(8)* (2004), 879–883.

[14] BEETZ, J., VAN LEEUWEN, J. P., AND DE VRIES, B. An Ontology Web Language notation of the Industry Foundation Classes. In *22nd CIB W78 Conference on Information Technology in Construction, CIB-W78, Dresden, Germany* (2005).

[15] BERNERS-LEE, T., HENDLER, J., AND LASSILA, O. The semantic web. In *Scientific American Magazine*. Gerard Piel, 2001.

[16] BOCK, D., LAU, M., SCHÖNHERR, T., WOLLENHAUPT, B., HERDRICH, G., AND RÖSER, H.-P. PERSEUS - In-Orbit Validation for Electric Propulsion Systems TALOS and SIMP-LEX. In *27th International Symposium on Space Technology and Science, Tsukuba, Japan* (2009).

[17] BODDY, S., REZGUI, Y., COOPER, G., AND WETHERILL, M. Computer integrated construction: A review and proposals for future direction. *Advances in Engineering Software 38(10)* (2007), 677–687.

[18] BÖHNKE, D. Erstellung einer Flugzeugentwurfssprache für die aerodynamische Analyse mittels CFD-Methoden. Studienarbeit, Institut für Aerodynamik und Gasdynamik der Universität Stuttgart und am Institut für Statik und Dynamik der Universität Stuttgart, Januar 2009.

[19] BÖHNKE, D., REICHWEIN, A., AND RUDOLPH, S. Design Language for Airplane Geometries Using the Unified Modeling Language. In *ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE, August 30-September 2, San Diego, USA* (2009).

[20] BOOCH, G. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, 1994.

[21] BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. *The Unified Modeling Language User Guide* . Addison-Wesley, 1998.

[22] BUILDINGSMART INTERNATIONAL. Industry Foundation Classes (IFC) specification. IFC2x Edition 3, `http://www.iai-tech.org/products/ifc_specification/ifc-releases`, 2006.

[23] BURMESTER, S., GIESE, H., MÜNCH, E., OBERSCHELP, O., KLEIN, F., AND SCHEIDELER, P. Tool support for the design of self-optimizing mechatronic multi-agent systems. *International Journal on Software Tools for Technology Transfer 10(3)* (2008), 207–222.

[24] BURMESTER, S., AND TICHY, H. G. M. Model-Driven Development of Reconfigurable Mechatronic Systems with MECHATRONIC UML. *Model Driven Architecture, European MDAWorkshops: Foundations andApplications,MDAFA2003 andMDAFA2004 Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers* (2005), 47–61.

[25] CHANG, X., AND TERPENNY, J. Ontology-based data integration and decision support for product e-Design. *Robotics and Computer-Integrated Manufacturing 25(6)* (2009), 863–870.

[26] CHIN, K.-S., ZHAO, Y., AND MOK, C. STEP-Based Multiview Integrated Product Modelling for Concurrent Engineering. *International Journal of Advanced Manufacturing Technology 20(12)* (2007), 896–906.

[27] CIOCOIU, M., NAU, D. S., AND GRUNINGER, M. Ontologies for Integrating Engineering Applications. *Journal of Computing and Information Science in Engineering 1(1)* (2001), 12–22.

[28] COLEMAN, D., ARNOLD, P., BODOFF, S., DOLLIN, C., GILCHRIST, H., HAYES, F., AND JEREMAES, P. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[29] CONRAD, J., TILL, D., KÖHLER, C., WANKE, S., AND WEBER, C. Comparison of knowledge representation in PDM and by semantic networks. In *16th International Conference on Engineering Design ; 28 - 30 August 2007, Paris, France* (2007).

[30] CRANEFIELD, S., AND PURVIS, M. UML as an Ontology Modelling Language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence* (1999).

[31] DAMJANOVIC, V., BEHRENDT, W., PLÖSSNIG, M., AND HOLZAPFEL, M. Developing Ontologies for Collaborative Engineering in Mechatronics. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007. Proceedings* (2007).

[32] DEMARCO, T. *Structured Analysis and System Specification*. Prentice Hall, 1975.

[33] DIJKSTRA, E. W. Go To Statement Considered Harmful. *Communications of the ACM 11(3)* (1968), 147–148.

[34] DOUGLASS, B. P. *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley Professional, 2004.

[35] DREVERMAN, M. Adoption of Product model data standards in the Process Industry. Master's thesis, USPI-NL and Dept. of Technology Management, Eindhoven University of Technology, 2005.

[36] DU, X., JIAO, J., AND TSENG, M. M. Graph Grammar Based Product Family Modeling. *Concurrent Engineering 10(2)* (2002), 113–128.

[37] EASTMAN, C., AND AUGENBROE, F. Product modeling strategies for today and the future. In *Björk B-C and Jägbeck A. (eds.) The life-cycle of IT innovations in construction - Technology transfer from research to practice, Proc. CIB W78 conference, June 3-5 1998 Stockholm, Royal Institute of Technology.* (1998).

[38] EASTMAN, C., WANG, F., YOU, S.-J., AND YANG, D. Deployment of an AEC industry sector product model. *Computer-Aided Design 37(12)* (2005), 1214–1228.

[39] EASTMAN, C. M., AND FERESHETIAN, N. Information models for use in product design: a comparison. *Computer-Aided Design 26(7)* (1994), 551–572.

[40] EUZENAT, J., AND SHVAIKO, P. *Ontology Matching*. Springer, 2006.

[41] EYNARD, B., GALLET, T., NOWAKA, P., AND ROUCOULES, L. UML based specifications of PDM product structure and workflow. *Computers in Industry 55(3)* (2004), 301–316.

[42] FELFERNIG, A., FRIEDRICH, G., AND JANNACH, D. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering 15(2)* (2001), 165–176.

[43] FIORENTINI, X., RACHURI, S., MANI, M., FENVES, S. J., AND SRIRAM, R. D. An Evaluation of Description Logic for the Development of Product Models. Tech. rep., National Institute of Standards and Technology, NISTIR 748., 2008.

[44] FRIEDENTHAL, S., MOORE, A., AND STEINER, R. *A practical guide to SysML: The Systems Modeling Language*. Morgan Kaufmann OMG Press, 2008.

[45] FRITZSON, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley & Sons, 2004.

[46] GALLAHER, M. P., O'CONNOR, A. C., AND PHELPS, T. Economic Impact Assessment of International Standard for the Exchange of Product Model Data (STEP) in Transportation Equipment Industries. Tech. rep., NIST Planning Report 02-5, 2002.

[47] GAMMA, E., HELM, R., AND JOHNSON, R. E. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1995.

[48] GASEVIC, D., DJURIC, D., AND DEVEDZIC, V. *Model Driven Engineering and Ontology Development*. Springer, 2009.

[49] GIEHLING, W. An assessment of the current state of product data technologies. *Computer-Aided Design 40* (2008), 750–759.

[50] GIMÉNEZ, D. M., VEGETTI, M., AND HENNING, H. P. L. G. P. PRoduct ONTOlogy: Defining product-related concepts for logistics planning activities. *Computers in Industry 59(2-3)* (2008), 231–241.

[51] GRABOWSKI, H., ANDERL, R., AND POLLY, A. *Integriertes Produktmodell*. Beuth, 1993.

[52] GRAPE, P., AND WALDÉN, K. Automating the development of syntax tree generators for an evolving language. *Proceedings of the eighth international conference on Technology of object oriented languages and systems, Santa Barbara, California* (1992), 185–195.

[53] GROSS, J. Entwicklung eines UML-Modells zur Datenintegration beim digitalen Entwurf von Kleinsatelliten. Diplomarbeit, Institut für Raumfahrtsysteme der Universität Stuttgart, Dezember 2008.

[54] GROSS, J., REICHWEIN, A., RUDOLPH, S., BOCK, D., AND LAUFER, R. An Executable Unified Product Model Based on UML to Support Satellite Design. In *AIAA SPACE Conference & Exposition 14 - 17 September, Pasadena, California* (2009).

[55] GRUBER, T. R. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition 5(2)* (1993), 199–220.

[56] GRUHN, V., PIEPER, D., AND RÖTTGERS, C. *MDA: Effektives Software-Engineering mit UML2 und Eclipse*. Springer-Verlag Berlin Heidelberg, 2006.

[57] GU, P., AND CHAN, K. Product modelling using STEP. *Computer-Aided Design 27(3)* (1995), 163–179.

[58] HAN, K. H., AND DO, N. An object-oriented conceptual model of a collaborative product development management (CPDM) system. *International Journal of Advanced Manufacturing Technology 28 (7/8)* (2006), 827–838.

[59] HAREL, D., AND RUMPE, B. Meaningful modeling: What's the semantics of "semantics"? *Computer 37(10)* (2004), 64–72.

[60] HENDERSON-SELLERS, B., AND BULTHUIS, A. *Object-Oriented Metamethods*. Springer, 1997.

[61] ILAL, M. E. The quest for integrated design system: A brief survey of past and current efforts. *Middle East Technical University Journal of the Faculty of Architecture 24* (2007), 149–158.

[62] INSTITUTE, T. S. C. CIMsteel Integration Standards Release 2: Second Edition. `http://www.cis2.org`, 2003.

[63] ISO 10303-11:2004. Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESS language reference manual, 2004.

[64] ISO 10303-1:1994. Industrial automation systems and integration – Product data representation and exchange – Part 1: Overview and fundamental principles, 1994.

[65] ISO 10303-21:2002. Industrial automation systems and integration – Product data representation and exchange – Part 21: Implementation methods: Clear text encoding of the exchange structure, 2002.

[66] ISO 10303-214:2003. Industrial automation systems and integration – Product data representation and exchange – Part 214: Application protocol: Core data for automotive mechanical design processes, 2003.

[67] ISO 10303-221:2007. Industrial automation systems and integration – Product data representation and exchange – Part 221: Application protocol: Functional data and their schematic representation for process plants, 2007.

[68] ISO 10303-22:1998. Industrial automation systems and integration – Product data representation and exchange – Part 22: Implementation methods: Standard data access interface, 1998.

[69] ISO 10303-225:1999. Industrial automation systems and integration – Product data representation and exchange – Part 225: Application protocol: Building elements using explicit shape representation, 1999.

[70] ISO 10303-227:2005. Industrial automation systems and integration – Product data representation and exchange – Part 227: Application protocol: Plant spatial configuration, 2005.

[71] ISO 10303-41:1994. Industrial automation systems and integration – Product data representation and exchange – Part 41: Integrated generic resource: Fundamentals of product description and support, 1994.

[72] ISO 10303-42:1994. Industrial automation systems and integration – Product data representation and exchange – Part 42: Integrated generic resources: Geometric and topological representation, 1994.

[73] ISO 15926-1:2004. Industrial automation systems and integration – Integration of life-cycle data for process plants including oil and gas production facilities – Part 1: Overview and fundamental principles, 2004.

[74] ISO/IEC 19501:2005. Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2, 2005.

[75] ISO/IEC 19502:2005. Information technology – Meta Object Facility (MOF), 2005.

[76] JACOBSON, I., BOOCH, G., , AND RUMBAUGH, J. *The Unified Software Development Process*. Addison-Wesley Professional, 1999.

[77] JACOBSON, I., CHRISTERSON, M., JONSSON, P., AND ÖVERGAARD, G. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, 1992.

[78] JIAO, J. R., SIMPSON, T. W., AND SIDDIQUE, Z. Product family design and platform-based product development: a state-of-the-art review. *Journal of Intelligent Manufacturing 18(1)* (2007), 5–29.

[79] JOHANSSON, G., AND DETTERFELT, J. A UML based modeling approach for multi domain system products. In *Proceedings of NordPLM'06, Göteborg* (2006).

[80] JOHNSON, T., PAREDIS, C., AND BURKHARDT, R. Integrating Models and Simulations of Continuous Dynamics into SysML. In *Proceedings of Modelica'08, Bielefeld* (2008).

[81] KAROL, A., LAHTELA, H., HÄNNINENA, R., HITCHCOCK, R., CHEN, Q., DAJKA, S., AND HAGSTRÖM, K. BSPro COM-Server-interoperability between software tools using industrial foundation classes. *Energy and Buildings 34(9)* (2002), 901–907.

[82] KEMMERER, S. STEP: The Grand Experience. SP939, National Institute of Standards and Technology, Gaithersburg, MD; `http://www.mel.nist.gov/msidlibrary/doc/stepbook.pdf`., 1999.

[83] KERZHNER, A. A., AND PAREDIS, C. J. J. Using domain specific languages to capture design synthesis knowledge for model-based systems engineering. Proceedings of the ASME 2009 IDETC/CIE Conference , San Diego, 2009.

[84] KIKO, K., AND ATKINSON, C. A Detailed Comparison of UML and OWL, REIHE INFORMATIK TR-2008-004. Tech. rep., University of Mannheim - Fakultät für Mathematik und Informatik - Lehrstuhl für Softwaretechnik, 2008.

[85] KIM, K.-Y., MANLEY, D. G., AND YANG, H. Ontology-based assembly design and information sharing for collaborative product development. *Computer-Aided Design 38(12)* (2006), 1233–1250.

[86] KLEINER, S., ANDERL, R., AND GRÄB, R. A collaborative design system for product data integration. *Journal of Engineering Design 14(4)* (2003), 421–428.

[87] KLEPPE, A., WARMER, J., AND BAST, W. *MDA Explained: The Model Driven Architecture*. Addison-Wesley Professional, 2003.

[88] KOBRYN, C. UML 2001: a standardization odyssey. *Communications of the ACM 42(10)* (1999), 29–37.

[89] KOBRYN, C. UML 3 and the future of modeling. *Journal of Software and System modeling 3(1)* (2004), 4–8.

[90] KOGUT, P., CRANEFIELD, S., HART, L., DUTRA, M., BACLAWSKI, K., KOKAR, M., AND SMITH, J. UML for ontology development. *The Knowledge Engineering Review 17(1)* (2002), 61–64.

[91] KRAUSE, F.-L., FRANKE, H.-J., AND GAUSEMEIER, J. *Innovationspotenziale in der Produktentwicklung*. Carl Hanser Verlag München Wien, 2007.

[92] KRAUSE, F.-L., AND KAUFMANN, U. Meta-Modelling for Interoperability in Product Design. *CIRP Annals - Manufacturing Technology 56(1)* (2007), 159–162.

[93] KRIMA, S., BARBAU, R., FIORENTINI, X., SUDARSAN, R., AND SRIRAM, R. D. OntoSTEP: OWL-DL Ontology for STEP, NISTIR 7561. Tech. rep., NIST, National Institute of Standards and Technology, 2009.

[94] LA ROCCA, G., KRAKERS, L., AND VAN TOOREN, M. Development of an ICAD Generative Model for Blended Wing Body Aircarft Design. In *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, 4-6 September, Atlanta, Georgia* (2002).

[95] LEAL, D. ISO 15926 Life cycle data for process plant: An overview. *Oil & Gas Science and Technology - Revue de l'IFP 60(4)* (2005), 629–637.

[96] LEE, K., CHIN, S., AND KIM, J. A core system for design information management using Industry Foundation Classes. *Computer-Aided Civil and Infrastructure Engineering 18(4)* (2003), 286–298.

[97] LIAN, C., AND GUODONG, J. Product modeling for multidisciplinary collaborative design. *International Journal of Advanced Manufacturing Technology 30 (7/8)* (2006), 589–600.

[98] LIANG, J., SHAH, J., D'SOUZA, R., URBAN, S., AYYASWAMY, K., HARTER, E., AND BLUHM, T. Synthesis of consolidated data schema for engineering analysis from multiple STEP application protocols. *Computer-Aided Design 31(7)* (1999), 429–447.

[99] LIN, H., AND HARDING, J. A manufacturing system engineering ontology model on the semantic web for inter-enterprise collaboration. *Computers in Industry 58(5)* (2007), 428–437.

[100] LOCKETT, H., BARTHOLOMEW, P., AND GALLOP, J. The Management of Product Data in an Integrated Aircraft Analysis Environment. *Journal of Computing and Information Science in Engineering 4(4)* (2004), 359–364.

[101] MÄNNISTÖ, T., PELTONEN, H., MARTIO, A., AND SULONEN, R. Modelling generic product structures in STEP. *Computer-Aided Design 30(14)* (1998), 1111–1118.

[102] MÄNNISTÖ, T., AND SULONEN, R. Evolution of Schema and Individuals of Configurable Products. In *Advances in Computer Modeling. ER' 99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling Paris, France, November 15-18* (1999).

[103] MELLOR, S. J., AND BALCER, M. J. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

[104] MEYER, B. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990.

[105] NANDA, J., SIMPSON, T. W., KUMARA, S. R. T., AND SHOOTER, S. B. A methodology for product family ontology development using formal concept analysis and web ontology language. *Journal of computing and information science in engineering 6(2)* (2006), 103–113.

[106] NASSI, I., AND SHNEIDERMAN, B. Flowchart techniques for structured programming. *ACM SIGPLAN Notices 8(8)* (1973), 12–26.

[107] NAUR, P., AND RANDELL, B. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. Tech. rep., Brussels, Scientific Affairs Division, NATO, 1969.

[108] NECHES, R., FIKES, R., FININ, T., GRUBER, T., PATIL, R., SENATOR, T., AND SWARTOUT, W. R. Enabling technology for knowledge sharing. *AI Magazine 12(3)* (1991), 16–36.

[109] NILES, I., AND PEASE, A. Towards a standard upper ontology. In *International Conference on Formal Ontology in Information Systems (FOIS), pages 2-9.* (2001).

[110] NYTSCH-GEUSEN, C., KLEMPIN, C., VON VOIGT, J. N., AND RÄDLER, J. Integration of CAAD, Thermal Building Simulation and CFD by Using The IFC Data Exchange Format. In *Eighth International IBPSA Conference, Eindhoven, Netherlands, 967-74, August 11-14* (2003).

[111] OH, Y., HUNG HAN, S., AND SUH, H. Mapping product structures between CAD and PDM systems using UML. *Computer-Aided Design 33(7)* (2001), 521–529.

[112] OMG. MDA Guide Version 1.0.1. omg/03-06-01, `http://www.omg.org/docs/omg/03-06-01.pdf`. Accessed 15 Mar 2009, 2003.

[113] OMG. Meta Object Facility (MOF) Core Specification Version 2.0. formal/06-01-01, `http://www.omg.org/docs/formal/06-01-01.pdf`., 2006.

[114] OMG. Object Constraint Language, Version 2.0. formal/2006-05-01, `http://www.omg.org/spec/OCL/2.0/PDF/`., 2006.

[115] OMG. UML Diagram Interchange Version 1.0. formal/2006-04-04, `http://www.omg.org/spec/UMLDI/1.0/PDF`, 2006.

[116] OMG. UML Profile for System on a Chip (SoC). formal/06-08-01, `http://www.omg.org/docs/formal/06-08-01.pdf`., 2006.

[117] OMG. MOF 2.0/XMI Mapping, Version 2.1.1. formal/2007-12-01, `http://www.omg.org/docs/formal/07-12-01.pdf`., 2007.

[118] OMG. SysML final adopted specification. ptc/06-05-04, `http://www.omg.org/cgi-bin/apps/doc?ptc/06-05-04.pdf`., 2007.

[119] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0. formal/2008-04-03, http://www.omg.org/docs/formal/08-04-03.pdf., 2008.

[120] OMG. UML Profile for MARTE, Beta 2. ptc/08-06-09, http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf., 2008.

[121] OMG. Unified Modeling Language (OMG UML) Infrastructure Specification, Version 2.2. formal/2009-02-04, http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/, 2009.

[122] OMG. Unified Modeling Language (OMG UML) Superstructure Specification, Version 2.2. formal/2009-02-02, http://www.omg.org/docs/formal/09-02-02.pdf., 2009.

[123] PAREDIS, C. J. J., DIAZ-CALDERON, A., SINHA, R., AND KHOSLA, P. K. Composable Models for Simulation-Based Design. *Engineering with Computers 17(2)* (2001), 112–128.

[124] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15(12)* (1972), 1053 – 1058.

[125] PARR, G. R., AND EDWARDS, R. Integrated Modular Avionics . *Air & Space Europe 1(2)* (1999), 72–75.

[126] PARUNAK, H. V. D., AND ODELL, J. Representing social structures in UML. *Agent-Oriented Software Engineering II, Lecture Notes in Computer Science, Springer 2222* (2002), 1–16.

[127] PATIL, L., DUTTA, D., AND SRIRAM, R. Ontology-Based Exchange of Product Data Semantics. *IEEE Transactions on Automation Science and Engineering 2(3)* (2005), 213–225.

[128] PAVEZ, L., Ed. *STEP Datenmodelle zur Simulation mechatronischer Systeme*. Karlsruhe : Forschungszentrum Karlsruhe Technik und Umwelt, 2001.

[129] PEAK, R. S., BURKHART, R. M., FRIEDENTHAL, S. A., WILSON, M. W., BAJAJ, M., AND KIM, I. Simulation-Based Design Using SysML Part 1: A Parametrics Primer. In *17th International Symposium of the International Council on Systems Engineering, San Diego, California, USA June 24 -28.* (2007).

[130] PEAK, R. S., BURKHART, R. M., FRIEDENTHAL, S. A., WILSON, M. W., BAJAJ, M., AND KIM, I. Simulation-Based Design Using SysML Part 2: Celebrating Diversity by Example. In *17th International Symposium of the International Council on Systems Engineering, San Diego, California, USA June 24 -28.* (2007).

[131] PEAK, R. S., LUBELL, J., SRINIVASAN, V., AND WATERBURY, S. C. STEP, XML, and UML: Complimentary Technologies. *Journal of Computing and Information Science in Engineering 59* (2004), 1–20.

[132] POP, A., AKHVLEDIANI, D., AND FRITZSON, P. Towards unified system modeling with the ModelicaML UML profile. In *Proceedings of EOOLT'07, Berlin* (2007).

[133] PORTNER, P. H. *What is Meaning?: Fundamentals of Formal Semantics*. Wiley-Blackwell, 2005.

[134] RACHURI, S., HAN, Y.-H., FENG, S. C., ROY, U., WANG, F., SRIRAM, R. D., AND LYONS, K. W. Object-oriented representation of electro-mechanical assemblies using UML, NISTIR 7057. Tech. rep., NIST, Gaithersburg, MD, 2003.

[135] RACHURI, S., SUBRAHMANIAN, E., BOURAS, A., FENVES, S., FOUFOU, S., AND SRIRAM, R. D. Information sharing and exchange in the context of product lifecycle management: Role of standards. *Computer-Aided Design 40* (2008), 789–800.

[136] REDDY, S. Y., KENNETH, AND FERTIG, K. W. Design Sheet: A System for Exploring Design Space - Application to Automotive Drive Train Life Analysis. In *Proc. Artificial Intelligence in Design* (1996), Kluwer Academic Publishers, pp. 34736–6.

[137] REICHEL, R., ARMBRUSTER, M., TJADEN, H., ZIMMER, E., SPIEGELBERG, G., AND SULZMANN, A. Safety Critical Control Platform for Automotive Vehicles. In *6th Braunschweig Conference AAET Automation, Assistance and Embedded Real Time Platforms for Transportation, Braunschweig* (2005).

[138] RENSSEN, A. V. *Gellish: A generic Extensible Ontological Language*. IOS Press, 2005.

[139] REZGUI, Y., BODDY, S., WETHERILL, M., AND COOPER, G. Past, present and future of information and knowledge sharing in the construction industry: Towards semantic service-based e-construction? *Computer-Aided Design (in Press)* (2009), doi:10.1016/j.cad.2009.06.005.

[140] ROY, P. V., AND HARIDI, S. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[141] RUDOLPH, S. Know-How Reuse in the Conceptual Design Phase of Complex Engineering Products - Or: Are you still constructing manually or do you already generate automatically? In *Conference Proceedings Integrated Design and Manufacture in Mechanical Engineering (IDMME 2006), May 17-19th, Grenoble, France.* (2006).

[142] RUDOLPH, S., AND BÖLLING, M. Constraint-based conceptual design and automated sensitivity analysis for airship concept studies. *Aerospace Science and Technology 8(4)* (2004), 333–345.

[143] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. *Object Oriented Modeling and Design*. Prentice-Hall, 1991.

[144] SCHAEFER, J., AND RUDOLPH, S. Satellite design by design grammars. *Aerospace Science and Technology 9(1)* (2005), 81–91.

[145] SCHATTKOWSKY, T., HAUSMANN, J. H., AND ENGELS, G. Using UML Activities for System-on-Chip Design and Synthesis. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), LNCS, Springer* (2006).

[146] SIDDIQUE, Z., AND ROSEN, D. W. Product Platform Design: a Graph Grammar Approach. In *ASME Design Engineering Technical Conferences, September 12-16, 1999, Las Vegas, Nevada* (1999).

[147] SOININEN, T., TIIHONEN, J., MÄNNISTÖ, T., AND SULONEN, R. Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12(4)* (1998), 357–372.

[148] SONG, H., EYNARD, B., ROUCOULES, L., LAFON, P., AND CHARLES, S. Beyond geometric CAD system: implementation of STEP translator for multiple-views product modeller. *International Journal of Product Lifecycle Management 2(1)* (2007), 1 – 17.

[149] SOWA, J. *Knowledge Representation: Logical, Philosophical and Computational Foundations*. Thomson Learning, 1999.

[150] SOWA, J. F. The Challenge of Knowledge Soup. In *Episteme-1 Conference in Goa, India, in December 2004. Research Trends in Science, Technology and Mathematics Education, edited by J. Ramadas & S. Chunawala, Homi Bhabha Centre, Mumbai.* (2006).

[151] SRINIVASAN, V. Standardizing the specification, verification, and exchange of product geometry: Research, status and trends. *Computer-Aided Design 40* (2008), 738–749.

[152] SRINIVASAN, V. An integration framework for product lifecycle management. *Computer-Aided Design (in Press)* (2009), doi:10.1016/j.cad.2008.12.001.

[153] STARZYK, D. STEP and OMG Product Data Management specifications: A guide for decision makers. OMG Document mfg/99-10-04 and PDES Inc. Document MG001.04.00; `http://www.omg.org/cgi-bin/doc?mfg/99-10-04`. Accessed 15 Mar 2009, 1999.

[154] STOKES, M., Ed. *Managing Engineering Knowledge, MOKA: Methodology for Knowledge Based Engineering Applications*. Professional Engineering Publishing. LondonBury St Edmunds, 2001.

[155] SUDARSAN, R., FENVES, S., SRIRAM, R., AND WANG, F. A product information modeling framework for product lifecycle management. *Computer-Aided Design 37* (2005), 1399–1411.

[156] SZYKMAN, S., FENVES, S. J., KEIROUZ, W., AND SHOOTER, S. B. A foundation for interoperability in next-generation product development systems. *Computer-Aided Design 33(7)* (2001), 545–559.

[157] SZYKMAN, S., SRIRAM, R. D., AND REGLI, W. C. The Role of Knowledge in Next-generation Product Development Systems. *Journal of Computing and Information Science in Engineering 1(1)* (2001), 3–11.

[158] THIMM, G., LEE, S., AND MA, Y. Towards unified modelling of product lifecycles. *Computers in Industry 57(4)* (2006), 331–341.

[159] TURKI, S., AND SORIANO, T. A SysML extension for bond graphs support. In *IEEE ICTA'05, Thessaloniki* (2005).

[160] ULRICH, K. The role of product architecture in the manufacturing firm. *Research Policy 24(3)* (1995), 419–440.

[161] UNGERER, M., AND BUCHANAN, K. Usage Guide for the STEP PDM Schema V1.2. PDM Implementor Forum, Release 4.3, January 2002, `http://www.steptools.com/support/stdev_docs/express/pdm/pdmug_release4_3.pdf`. Accessed 15 Mar 2009, 2002.

[162] US PRO. Initial Graphics Exchange Specification IGES 5.3. US Product Data Association (USPRO); `http://www.uspro.org/documents/IGES5-3_forDownload.pdf`., 1996.

[163] VERBAND DER AUTOMOBILINDUSTRIE E.V (VDA). VDA Surface Interface Version 2.0, 1987.

[164] W3C. OWL Web Ontology Language Guide. W3C Recommendation 10 February 2004, `http://www.w3.org/TR/owl-guide/`, 2004.

[165] WEB3D. The Virtual Reality Modeling Language VRML97. International Standard ISO/IEC 14772-1:1997, `http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/`, 1997.

[166] WEB3D. Extensible 3D (X3D). ISO/IEC FDIS 19775-1:2008, `http://www.web3d.org/x3d/specifications/ISO-IEC-19775-1.2-X3D-AbstractSpecification/index.html`, 2008.

[167] WEGNER, P. Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger 1(1)* (1990), 7 – 87.

[168] WITHERELL, P., KRISHNAMURTY, S., AND GROSSE, I. R. Ontologies for Supporting Engineering Design Optimization. *Journal of Computing and Information Science in Engineering 7(2)* (2007), 141–150.

[169] YANG, D., DONG, M., AND MIAO, R. Development of a product configuration system with an ontology-based approach. *Computer-Aided Design 40(8)* (2008), 863–878.

[170] YANG, Q., AND ZHANG, Y. Semantic interoperability in building design: Methods and tools. *Computer-Aided Design 38(10)* (2006), 1099–1112.

[171] ZHA, X. F., AND DU, H. A PDES/STEP-based model and system for concurrent integrated design and assembly planning. *Computer-Aided Design 34(14)* (2002), 1087–1110.

[172] ZHOU, Z. D., XIE, S. Q., AND YANG, W. Z. A case study on STEP-enabled generic product modelling framework. *International Journal of Computer Integrated Manufacturing 21(1)* (2008), 43–61.

[173] ZOUGHBI, G., BRIAND, L., AND LABICHE, Y. A UML Profile for Developing Airworthiness-Compliant (RTCA DO-178B), Safety-Critical Software. In *10th International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS, Springer* (2007).