VALIDATING AND APPLYING MODEL TRANSFORMATIONS

László Lengyel

Dissertation submitted
for the degree of Doctor of the Hungarian Academy of Sciences

Budapest, 2018

Contents

List of Figures

Summary

Software is a must to have artifact. We are continuously developing applications for every aspect of our life, for business issues, for various large-scale, embedded and smart devices as well. We use different development methods to support these activities. Applications and services continuously generate huge data streams especially when new sensors, mobile devices, smart solutions and different modern tools are considered. Storing, processing, analyzing, extracting actionable information and utilizing this data requires domain knowledge, algorithms, processes, effective methods and powerful infrastructure. The research results discussed in the thesis are supporting these activities by methods providing effective system design and development. The core motivation is to utilize domain-specific modeling and model processing to improve the quality of the model processors, and therefore, the quality of the generated software artifacts.

The growing dimension and complexity of software systems have turned software modeling technologies and model-driven development into an efficient tool in application development. Within the modeling approaches, there exists a clear trend to move from universal modeling languages towards domain-specific solutions. Domain-specific languages are strictly limited to a domain, but this limitation also makes them much more efficient. The motivation behind domain-specific modeling is to understand the rules and processes of the organization/domain that we are about to support with software artifacts. Further goal is to understand the actual tasks and challenges of the organization, define them as domain models, and derive the software system related requirements from these models. The focus point of the research activities are to work out and apply methods for the following areas:

- *Provide domain-specific methods to support effective requirements engineering, specification, software modeling, model processing, i.e. development and maintenance.*
- *Work out methods for verifying and validating model processors to ensure high quality software artifacts.*
- *Apply domain-specific design patterns and validate domain-specific properties.*

Research directions are influenced by two main aspects. The first aspect is to follow the international research trends, be an active and determining part of the community, furthermore, from time to time, contribute outstanding results on certain areas. The second aspect is to support model-driven development related industrial requirements. I believe that the real value of research results manifests in their application and utilization. Therefore, the selection of research directions and working on them have always been significantly affected by the strategies, goals and requirements of the application area.

The thesis emphasizes the necessity of domain-specific tools, and methods that make development activities validated, discusses the different scenarios of model transformation verification and validation, furthermore, introduces the principles of several novel model-driven methods and techniques for validating domain-specific system properties.

Összefoglaló

A szoftver, legyen szó szolgáltatásokról vagy alkalmazásokról, a mindennapjaink része. Folyamatosan fejlesztünk megoldásokat az élet változatos részeinek támogatására, különféle komplex irányításokra és vezérlésekre, beágyazott rendszerekbe és okos eszközökre. Változatos módszereket és fejlesztőeszközöket alkalmazunk és használunk ezen tevékenység során. Az alkalmazások és szolgáltatások, különösen igaz ez nagyszámú szenzorral felszerelt rendszerek esetén, hatalmas adatmennyiséget generálnak. Ezen adatok tárolása, feldolgozása, elemzése, a tettre fogható információ kinyerése és hasznosítása szakterületi tudást, hatékony algoritmusokat, folyamatokat, módszereket, valamint megbízható infrastruktúrát igényel. Az értekezésben tárgyalt kutatási eredmények ezen célokhoz járulnak hozzá a rendszertervezést és a fejlesztést támogató módszerekkel. A kutatómunka meghatározó motivációja a modellfeldolgozók, ezáltal a szoftvertermékek, minőségének növelése, melynek központi eszközrendszere a szakterületi modellezés és modellvezérelt technikák alkalmazása.

A szoftverrendszerek növekvő komplexitása révén kerül előtérbe a modellvezérelt fejlesztés, mint hatékony eszköz. Meghatározó eleme a szakterületi modellezés alkalmazása, melynek kiemelt célja megérteni annak a szervezetnek és alkalmazási területnek a felépítését és működését, amelynek a munkáját szoftvertermékekkel támogatni fogjuk. Cél megérteni a szervezet aktuális feladatait, szakterületi modellek formájában rögzíteni, valamint származtatni a szoftverrendszerhez kapcsolódó követelményeket. A kutatómunka célkitűzéseinek központi elemei a következők:

- *A követelményelemzés, a szoftvermodellezés és a modellfeldolgozás támogatása, módszerek kidolgozása a szakterület-specifikus elemek figyelembe vételével és alkalmazásával.*
- *A modellfeldolgozók helyességvizsgálatát támogató módszerek és megoldások kidolgozása és alkalmazása.*
- *Szakterület-specifikus tulajdonságok validálását és szakterület-specifikus tervezési minták alkalmazását támogató módszerek kidolgozása és használata.*

A kutatási irányokat folyamatosan két fő szempont befolyásolta. Az első tényező a nemzetközi kutatási trendek követése, szerepvállalás és bizonyos területeken kiemelkedő eredmények és teljesítmény felmutatása. A második tényező a modellvezérelt fejlesztéshez kapcsolódó ipari igények támogatása. Fontos pontnak tartom annak felismerését, hogy a tudományos eredmények valódi értéke az alkalmazásukban is megmutatkozik. Ezért a kutatási témák megválasztásában és művelésében mindig meghatározó szerepe volt az alkalmazói szféra stratégiájának, céljainak, valamint a tudományos műhelyünktől elvárt eredményeknek.

Az értekezés hangsúlyosan foglalkozik a szakterületi eszközök szükségességével, a módszerek fontosságával, melyek a fejlesztési aktivitásokat, a modellfeldolgozókat hivatottak validálni. Tárgyalásra kerülnek a modellfeldolgozók validálásának különböző forgatókönyvei, valamint több új módszer bevezetése történik meg a szoftverrendszerek szakterülti tulajdonságainak garantálására és a modellfeldolgozók validálására.

Acknowledgements

This thesis could not have been created without the support of many people. First of all, I am grateful to my family, because of their support, tolerance and understanding. Furthermore, I would like to thank my friends the time spent together also helped my work.

I am indebted to Tihamér Levendovszky, who inspired my research activities during the early period. I would like to thank him the long Friday afternoon consultations, and later the skype discussions. He has motivated and guided me through this endeavor. This work could not have happened without him.

I am indebted to Hassan Charaf for providing the power and the human conditions of the work, furthermore, his advices related both to research directions and not research related topics. I would like to thank to István Vajk and Jenő Hetthéssy their stimulating words and advices.

I am grateful to Gergely Mezei for the common work in the implementation, and I would like to thank him his useful research related questions and remarks. I would like to thank the colleges at the Department of Automation and Applied Informatics (Budapest University of Technology and Economics) their help. I also thank my coauthors for the common work.

Thanks to the reviewers of the papers their very useful advices, criticism, suggestions, remarks and questions. The reviews helped me a lot during the work.

Finally, I would like to give thanks to God bringing all these people into my life.

dc_1522_18

1 Introduction

The information and communication technologies (ICT) play a horizontal role both in the society and in the economy. ICT has a carrier role; it significantly contributes to the competitiveness of various domains. Based on the industry-defined requirements, ICT supports the rapid application and utilization of various research results in various domains. Our society requires more and more high-quality applications and services. This motivates the ICT sector to work out convergent development methodologies and provide sustainable development processes. We are continuously developing applications for every aspect of our life, for the business issues and for different large scale, embedded and smart devices as well. We use different development methods to support these activities. To increase the effectiveness of development and improve the quality of software artifacts, we apply model-driven methods, i.e. we move the design to higher abstraction level and derive source code, configuration files and further artifacts from software models.

Model-driven software engineering is a discipline in software engineering that relies on models as first-class artifacts that aim to develop, maintain, and evolve existing software through the implementation of model transformations. Model-driven software engineering approaches emphasize the use of models at all stages of system development. As the necessity for reliable systems increases, both the specification of model transformations and the verification and validation of model transformation-based approaches become emerging research fields. In this context, verification and validation mean determining the accuracy of a model transformation and ensure that the output models of the transformation satisfy certain conditions.

Model transformations appear in a variety of ways in the model-based development process [Sztipanovits et al, 1997] [Sendall and Kozaczynski, 2003] [Küster, 2006]. A few representative examples are as follows. (i) Refining the design to implementation [Barbosa, 2009] [6]; this is a basic case of mapping platform-independent models to platform-specific models. An example of this exists in the current MDA initiative [OMG MDA, 2014] which favors the use of model transformations, within UML-based [OMG UML, 2015] development of software systems, for a variety of different purposes. (ii) Transforming models into other domains, e.g., transforming system models into a mathematical domain (transition systems, Petri nets, process algebras, etc.) to perform a formal analysis of the system under design [Biermann et al, 2011] [Varró, 2004]. (iii) Aspect weaving; the integration of aspect models/code into functional artifacts is considered a transformation on the design [Assmann and Ludwig, 2000]. (iv) Analysis and verification: certain analysis algorithms can be expressed as transformations on the design [Assmann, 1996] [5]. (v) Refactoring purposes: improving model attributes and/or model structure while preserving the external semantic meaning [v. Gorp et al, 2003]. (vi) Simulation and execution of a model as operational semantics, migration, normalization and optimization of the models [Amrani et al, 2012] [Taentzer et al, 2005].

As model transformations are being applied to so many diverse scenarios, there is a compelling need for techniques and methodologies regarding their further development, and also for verifying/validating them [Cabot et al, 2010].

There are methods, techniques and tools successfully applied to develop robust, large-scale software systems. Their design, development, testing, operation and maintenance activities require reasonable time and resources. However, with the growing volume of required software systems, these activities should be optimized, better supported with methods and tools, in order to preserve or even increase the quality with optimized resource allocation.

Developing and then maintaining complex frameworks, e.g. AWS IoT [AWS IoT], Azure IoT Suite [Azure IoT Suite] or SensorHUB [9], furthermore, designing/developing services and applications on top of such extensive platforms, requires convergent development methods, appropriate tool support, furthermore, effective management and application of architectural patterns, design patterns and best practices. These elements, i.e. software products related patterns and best practices are basically domain-specific or they are related to domain-specific rules and processes.

As a conclusion, we have found that the growing dimension and complexity of software systems have turned software modeling technologies and domain-specific methods into efficient tools in application design and development, i.e. during the whole process: requirements engineering and analysis, specification, design, development, testing, documentation and maintenance.

We aim for model properties, for example in requirements' models, important for the assessment, to be transformed precisely into the output domain (software systems). During the design of such a transformation, and later during the application of this transformation, we face the following questions: What ensures that the transformation process to the output domain is correct? What type of properties can a transformation preserve?

1.1 Motivations

We often project models into another domains or formats, for example, into formal models. In addition, we always ask, what ensures that the projection is free of conceptual errors? The central question of the area is the following: how can we ensure that the model transformation does what it is intended to do?

For most software systems, the design process requires the continuous validation of the design decisions. Modeling languages and general modelling tools control the syntactic of the models but could not guarantee the correctness of the design. Therefore, during the design process of digital systems, we often transform design models into various formal domains in order to perform analysis of the system under design. These model transformations between the different representations should preserve key semantic properties of the software models.

The goal of the transformations' analysis is to show that in case of valid input models, certain properties will be true for the output model. The analysis of a transformation is said to be *static* when the implementation of the transformation and the language definition of the input and output models are used during the analysis process, but we do not take concrete input models into account. In the case of the *dynamic* approach, we analyze the transformation for a specific input model, and then check whether certain properties hold for the output model during or after the successful application of the transformation. The static technique is more general and poses the more complex challenges.

In order to further strengthen the motivation of the research area, the following paragraphs provide a collection of challenging transformations defined by different research groups.

[Giese et al, 2006] discussed that the problem in using model-driven software development (MDD) is the lack of verified transformations, especially in the area of safety-critical systems. The verification of crucial safety properties on the model level is only useful if the automatic code generation is guaranteed to be correct, i.e., the verified properties are guaranteed to hold for the generated code as well. This means it is necessary to pay special attention to the checking of semantic equivalence, at least to a moderate level, between the model specification and the generated code.

In the field of developing safety-critical systems, model analysis possesses advantages over the pure testing of implemented systems. For example, required safety properties of a system under development could be verified on the model level rather than trying to systematically test for the absence of failures.

This and similar conditions require a guarantee that properties verified at the model level are transformed correctly into source code.

[Narayanan and Karsai, 2008] summarized that, in model-based software development, a complete design and analysis process involves designing the system using the design language, converting it into the analysis language, and performing the verification on the analysis model. They stated that graph transformations are a powerful and convenient method increasingly being used to automate this conversion. In such a scenario, the transformation must ensure that the analysis model preserves the semantics of the design model. Important semantic information can easily be lost or misinterpreted in a complex transformation due to errors in the graph rewriting rules or in the processing of the transformation. They concluded that methods are required to verify that the semantics used during the analysis are indeed preserved across the transformation.

[de Lara and Taentzer, 2004] discussed the need for verified and validated model processing in the field of Multi-Paradigm Modeling (MPM) [de Lara et al, 2004]. Software systems have components that may require descriptions using different notations, due to different characteristics. For the analysis of certain properties of the whole system, or its simulation, we transformed each component into a common single formalism, in which appropriate analysis or simulation techniques are available.

A similar situation arises with object-oriented systems described in UML, where various views of the system are described through different diagrams. For the analysis of such a system, the different diagrams can be translated into a common semantic domain. These and similar model transformations should ensure the preservation of relevant system properties.

[de Lara and Guerra, 2009] provided formal semantics for QVT-Relations (Query, Views, Transformations) [OMG QVT, 2016], through the compilation into Colored Petri nets (CPNs), enabling the execution and validation of QVT specifications. The theory of Petri nets provides useful techniques to analyze transformations (e.g. reachability, model checking, boundedness and invariants) and to determine their confluence and termination, given a starting model. This approach requires that transformations, converting QVT-Relations models into CPNs, preserve the semantics relevant to the analysis.

[Varró, and Pataricza, 2003] states that, for most computer-controlled systems, an effective design process requires an early validation of the concepts and architectural choice. Therefore, during the design of systems, models are frequently projected into various mathematical domains (such as Petri nets, process algebras, etc.) in order to perform formal analysis via automatic model transformations. Automation certainly increases the quality of such transformations as errors manually implanted into transformation programs during implementation are eliminated. Consequently, verification and validation of model transformations is required, which assures that conceptual flaws in transformation design do not remain undetected.

[Varró, 2004] went on to state that, due to the increasing complexity of IT systems and modeling languages, conceptual, human design errors will occur in any models on any high-level and even of the formal modeling paradigm. Accordingly, the use of formal specification techniques alone does not guarantee the functional correctness and consistency of the system under design. Therefore, automated formal verification tools are required to verify the requirements fulfilled by the system model. As the input language of model checker tools is too basic for direct use, model transformations are applied to project behavioral models into the input languages of the model-checking tools.

To summarize, it is crucial to understand that model transformations themselves can be erroneous; therefore, uncovering solutions to make model transformations free of conceptual errors is necessary.

1.2 Structure of the Thesis

The Thesis has 7 chapters which are organized in the following way:

- Chapter 1 has provided the introduction, motivations and the main objectives related to the research activities.
- Chapter 2 is the state of the art area, this chapter devoted to illustrate the research area: software modeling and model processing, model-driven development, verification and validation of model transformations.
- Chapter 3 discusses the novel methods for verifying and validating graph rewriting-based model transformations. The chapter covers a suggested classification of model transformation approaches by model processing properties, a method for validating rule-based systems, suggestions for taming the complexity of model transformation verification/validation processes, furthermore, a method and algorithms to support test-driven verification/validation of model transformations.
- Chapter 4 suggests model-driven methods based on domain-specific languages and model processors. The chapter introduces a method for assuring the quality of software development projects with applying model-driven techniques and model-based tools, provides a method for developing and managing domain-specific models, a method for supporting the transparent switch between the textual and visual views of semantic models, a method for processing Mathworks Simulink models with graph rewriting-based model transformations, furthermore suggests a model-driven method for managing energy efficient operating properties.
- Chapter 5 provides methods for applying domain-specific design patterns and validating domain-specific properties of software models. The chapter discusses a method to support domain-specific design patterns, a method and algorithms for validating the domain-specific properties of software models, furthermore, a method and algorithms for handling the validating constraints in a modular way.
- Chapter 6 discusses the application fields of the achieved scientific results and introduces several applications that utilize the research results. Furthermore, some Research & Development projects are also introduced that utilized several elements of the results.
- Finally, Chapter 7 concludes the Thesis by summarizing the main scientific results.

2 Backgrounds

This section provides the state of the art overview of those information technology fields and research areas that significantly contribute to the overall goal of the thesis, i.e. the verification and validation of model transformations and the application of both domain-specific techniques and model-driven solutions.

Technology trends continue by the unstoppable path towards cloud computing, big data, applications, mobile devices, wearable gadgets, 3D printing, integrated ecosystems, and of course the Internet of Things (IoT) as the next computing platform [Swan 2013] [Thibodeau, 2014].

2.1 Internet of Things

The Internet of Things (IoT) is transforming the surrounding everyday physical objects into an ecosystem of information that enriches our everyday life. The IoT represents the convergence of advances in miniaturization, wireless connectivity and increased data storage and is driven by various sensors. Sensors detect and measure changes in position, temperature, light, and many others, furthermore, they are necessary to turn billions of objects into data-generating "things" that can report on their status, and often interact with their environment.

The goal of the Internet of Things (IoT) is to increase the connectedness of people and things. The IoT is the network of physical *things* equipped with electronics, software, sensors and connectivity that provides greater value and better service by exchanging data with the manufacturer, operator and/or other connected devices. Each element of the network, i.e. each thing, is uniquely identifiable through its embedded computing system and is able to interoperate within the existing Internet infrastructure.

Things in the IoT can refer to a wide variety of devices such as biochips on farm animals, heart monitoring implants, production line sensors in factories, vehicles with built-in sensors, or field operation devices that assist firefighters. These devices collect useful data with the help of various existing technologies, then autonomously flow the data between other devices and usually upload them into a cloud environment for further processing.

The IoT together with the collected and analyzed data can help consumers achieve goals by greatly improving their decision-making capacity via the augmented intelligence of the IoT. For businesses, the Internet of Business Things helps companies achieve enhanced process optimization and efficiency by collecting and reporting on data collected from the business environment. More and more businesses are adding sensors to people, places, processes and products to gather and analyze information in order to make better decisions and increase transparency.

Undoubtedly, the Internet of Things has reached and is about to dominate several domains. Top industries investing in sensors and utilizing data collected by them are as follows (some of them are still in active research phase, because of technical challenges and economic issues, but others are already being implemented) [Sensing IoT, 2015]:

- *Energy & Mining* – Sensors continuously monitor and detect dangerous carbon monoxide levels in mines to improve workplace safety.
- *Power & Utilities* – In the past, and mostly today, power usage is still measured on a yearly basis. However, Internet-connected smart meters can measure power usage every 15 minutes and provide feedback to the power consumer, often automatically adjusting the system's parameters.

- *Transportation and Vehicles* – Sensors planted on the roads, working together with vehicle-based sensors, are about to be used for hands-free driving, traffic pattern optimization and accident avoidance.
- *Industrial Internet (Industry 4.0)* – A manufacturing plant distributes plant monitoring and optimization tasks across several remote, interconnected control points. Specialists once needed to maintain, service and optimize distributed plant operations are no longer required to be physically present at the plant location, providing economies of scale. This is one of the areas where significant improvements are expected in the near future.
- *Hospitality and Healthcare* – Electronic doorbells silently scan rooms with infrared sensors to detect body heat, so the staff can clean when guests have left the room. Electro Cardio Graphy (ECG) sensors work together with patients' smartphones to monitor and transmit patients' physical environment and vital signs to a central cloud-based system.
- *Retail* – Product and shelf sensors collect data throughout the entire supply chain. They often provide from dock to shelf logs. Predictive analytics applications process these data and optimize the supply chain.
- *Technology* – Hardware manufacturers continue to innovate by embedding sensors to measure performance and predict maintenance needs.
- *Financial Services* – Telematics allows devices installed in the car to transmit data to drivers and insurers. Applications like stolen vehicle recovery, automatic crash notification, and vehicle data recording can minimize both direct and indirect costs while providing effective risk management.

Wearable devices such as activity trackers, smart watches and smart glasses are good examples of the Internet of Things (IoT), since they are part of the network of physical objects or *things* embedded with electronics, software, sensors and connectivity to enable objects to exchange data with a manufacturer, operator and other connected devices, without requiring human intervention.

Smart, connected products are products, assets and other things embedded with processors, sensors, software and connectivity that allow data to be collected, aggregated, exchanged between the product and its environment, manufacturer, operator or user, and further devices and systems. Connectivity also enables some capabilities of the product to exist outside the physical device, i.e. in the cloud.

Smart Sensors are context (vehicle, forest, city, water, others) and domain (transportation, climate, energy, smart city, smart building, others) aware and the collected data provide their real meaning, i.e. their semantics, within the given environmental and time dimensions.

According to the surveys and estimations [BBC Research, 2017] [Slash Data, 2017a, 2017b] [Vision Mobile, 2015a, 2015b], there will be 50 billion sensor-enabled objects connected to networks by 2020, and 212 billion will be the total number of available sensor-enabled objects by 2020. The latter one is 28 times the total population of the world. Further numbers by 2020 are: 4 billion connected people, 25+ million applications, 25+ billion embedded and intelligent systems, 50 trillion GBs of data.

There are two aspects of the IoT world: the first one is collecting data, processing and analyzing it; the second is providing services and applications on top of the analyzed data in order to support third-party services and serve end-users. Data monetization, i.e. controlled data sharing, is part of the second aspect, where well-defined slices (views) of the data represent valuable base information for different industrial sectors.

"Data is the new oil." [Humbly, 2006] We often meet similar statements. IoT-based data collection, data transmission, big data management, trusted cloud, and privacy issues are the main challenges of this area. Frameworks help companies and research groups to contribute to the IoT ecosystem as well as to the future design and to the development platforms. Based on the development results and ongoing project activities, we can state that SensorHUB (Section 6.1.2) worked out by our team, utilizing a part of the results summarized in this thesis, is such a framework.

2.2 Software Development Methodologies

Software industry and information technology methods are affected and shaped by end-user, domain-related and industrial requirements, trends and the fact that powerful hardware and communication infrastructure are widely available. The goal of the currently emerging ***software solution*** approaches is to address the values of unified development methods (targeting various devices and platforms, including cloud-based services), software-intensive and zero maintenance requirements, energy-efficient applications, cooperative behavior, software quality, lasting hardware and software solutions (e.g., sensors with their embedded software).

There are several areas and capabilities of the ICT ecosystem that shapes the development processes and have an effect on the elements and structures of the development methodologies.

Infrastructure, platform and software services are available in the cloud. These services are robust, reliable, secure, scalable, and are always available with huge storage and powerful processing capacity. Their availability is natural, we use and utilize them as a *public utility*.

We live in exponentially growing world, where *ICT has a determining position and has a horizontal role*, i.e. *responsible to make other domains competitive*. Novel methodologies are about to be *sustainable,* in order to make both development and maintenance efficient.

2.2.1 Integrated Solutions

There are given conditions and achieved results in both the hardware and the software areas, furthermore, their combination has determined the current integrated solutions.

The *hardware*-related field, with its continuous development, contributes valuable conditions. Some representative examples:

- Raspberry Pi zero, the 5 USD computer. Raspberry Pi is driving down the cost of computer hardware, i.e. the programmable computer.
- Average price of IP-enabled sensors will be only 2 USD within years.
- Usage-based cloud services dominate the ICT area. Examples of cloud services include online data storage and backup solutions, web-based e-mail services, hosted office suites and document collaboration services, push notification, database processing, managed technical support services and more.
- Huge storage and computing capacity (service) is available on reasonable price.
- Smart network is available: intelligent network solutions, i.e. routers, switches, network coding, given infrastructure elements and their software components.

On the *software* area, there are also several achievements that support effective development and related methodologies:

- Multi-platform development methodology [6]: a method, which increases the development productivity of the same functionality for various platforms and ensures the quality of applications.
- Several effective IDEs are available and ready to use. Examples are Eclipse, IDEA and Visual Studio.
- IoT, big data analyses, business intelligence, reporting and visualization frameworks to increase the productivity. Example frameworks are SensorHUB [9], AWS IoT [AWS Iot] and Azure IoT Suite [Azure IoT Suite].
- Unified high-level language for software design: OMG's UML [OMG UML, 2015].
- Domain-Specific Languages (DSLs) for dedicated domains to support the effective common work of domain experts and software architects.

These points are examples to underpin, that both hardware infrastructure and software conditions are developing, they are available for utilization, and *our further added value should be in our research and engineering capacities*. *Our next step is the capability that enables to realize real solutions in a sustainable way and provide real value for the affected domains.*

Integrated *solutions*, i.e. our present in the ICT field, are affected and labelled by these conditions. We can summarize that *software-intensive solutions dominate the ICT area and an increasing number of domains*. We are overwhelmed with a large number of applications. The digital enlightenment reaches a wider range of population, i.e. more and more people can reach and use ICT-enabled services. The wearable devices are about to conquer the near future. Autonomic computing, i.e. self-managing characteristics of distributed computing resources, with the capability to adapt to unpredictable changes also dominates the solutions area. Finally, the *cost of human resources is rather high in the ICT field.*

We see our *future in the solutions area*, which follows the further development of devices, e.g. biosensor-enabled smartphones, latency issues (5G) with software-intensive solutions, automatic updates in a pushed way. Furthermore, it *provides a transparent handling method over the diversity of devices, techniques, tools and methodologies*. The significance of the domain knowledge is increasing, which being combined with the common industrial requirements, enhances the weaving role of software solutions (*Weaving ICT*), i.e. existing and novel research results can rapidly be applied for various domains.

ICT companies see that business is shifting towards services. This development will naturally imply a future business with more recurring software and services revenues. Hardware components would always remain part of the solutions and will be one of the key differentiators. Companies now want to make money when people use their services, not when people buy their devices.

In this area, software and the capability to efficiently develop high-quality, sustainable services and applications have key role. Development processes require appropriate methodologies, tools and IT specialists.

2.2.2 Impacts of the Development Methodologies

Software development methodologies aim at four target groups of people, which can benefit from its results. For each of the groups below, we specify tangible impact objectives with example measures and justify the impact.

Software Developers

1. *Productivity*: Automation and shortening of cycle from requirements to code.

2. *Quality*: Repeatable code generated from precise requirements leave much less space for errors.
3. *Time to market*: Higher productivity and quality (see above) results in faster release of systems to the market.
4. *Methodology, best practices, patterns, examples to follow*: Defined, tested, documented methods tested and verified by several senior developers, architects and researchers.

Software Tool Vendors

1. *Tool offerings*: Methods and technologies exhibit valuable benefits for software developers which causes relevant prospects for related tool sales.
2. *Customer base*: Wider customer base is predicted due to effectiveness of the tools for domain experts and end-users.
3. *Market take-up*: Methods and technologies are disseminated and gain attention of external tool vendors due to validated benefits for software developers.

Software Users (Industry)

1. *Compliance with requirements*: Software generated from domain models better fulfils the end-user needs. The end-users are able to control this compliance in a direct way.
2. *Software acquisition*: Due to market competition, software developers shift their savings from increased productivity, in part to their clients.
3. *Software reuse*: UI-based reuse allows for high levels of recovery of application logic (use case scenarios) into domain models. Lower levels can be achieved for services due to their technology dependence.
4. *End-user involvement*: Project goal related results are developed from well-accepted standard notations, suitable for communications with domain experts.
5. *Software development framework*: Supporting the effective development for multi-platform environment in a unified way.

Software Engineering Educators and Researchers

1. *Methodology, best practices, patterns, examples to follow*: Defined, tested, documented methods tested and verified by several senior developers, architects and researchers.
2. *Course offerings*: Methods and techniques have high potential in terms of novelty and *coolness* to gain attention many software developers and end-users. This opens market for course offerings both in commerce and in academia.

Appropriate software development methodologies significantly reduce effort to formulate requirements and turn these requirements into working systems. Methods are about to adapt to the rapidly changing conditions with putting the domain requirements into the center. In summary, *development methodologies* promise *productivity and quality increase*. At the same time, novel methods are expected to cause significant *research community and industrial market take-up* of its innovative methods and thus contributing to *competitiveness and growth* of ICT research teams and software tool companies.

Based on the European ICT programs (H2020, Tools and Methods for Software Development), development methodologies realize the following contribution towards expected impacts. Model-based methods, domain-specific approaches, effective model-driven solutions highly contribute to these goals.

1. *A significant and substantiated productivity increase in the development, testing, verification, deployment and maintenance of data-intensive systems and highly distributed applications.*
    - Efficient supporting methodologies, development of software at the level of requirements, leveraging coarse grained reuse of available services.
    - Concentration on domain models which facilitate processing of data within data-intensive systems.
    - Automatic translation of requirements models into application and cloud-enabled service code ready for deployment in a selected highly distributed infrastructure.
    - Instant testing and verification against requirements, through executing and simulating code generated from these requirements.
    - Extending the base for productivity increase by direct involvement of end-users (domain experts) in the development of effective code.
2. *Availability and market take-up of innovative tools for handling complex software systems. A credible demonstration that larger and more complex problems can be effectively and securely tackled.*
    - The main objective of the program is to make *available* a set of *innovative methods, patterns and best practices* to handle complexity at the level of requirements. The methods and technologies used within these tools will be prepared for *easy take-up by the software tool market*, resulting in new innovative tool offerings.
    - Domain experts can be effectively involved in formulating requirements.
3. *At macro level, evidence of potential for productivity gains through appropriate use cases in the industry.*

2.3 Software Modeling and Domain-Specific Languages

Software modeling is a key concept, which supports requirements engineering, requirements analysis, facilitates the system definition and allows better communication on the appropriate abstraction level. Furthermore, system models are the first-class artifacts in model-based development. Modeling and model-based development gather several fields, such as UML [OMG UML, 2015], domain-specific modeling, multi-paradigm modeling [de Lara et al, 2004], generative programming [Czarnecki and Eisenecker, 2000] or model processing [Amrani et al, 2012] [Mens and v. Gorp, 2006] [Sendall and Kozaczynski, 2003].

The growing dimension and complexity of software systems have turned software modeling technologies into an efficient tool in application development. Within the modeling approaches, we are moving from universal modeling languages towards domain-specific solutions.

In software development and domain engineering, a domain-specific language (DSL) [Fowler, 2010] [Kelly and Tolvanen, 2008] is either a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. DSLs are strictly limited to a domain, but this limitation also makes them much more efficient than universal languages.

By *domain-specific languages,* we mean textual or visual languages that are used in a more specialized way than in general-purpose programming. DSLs have limited expressive potential and can only describe problems from a well-defined problem domain. These characteristics make them suitable to

achieve several different intents. Using domain-specific artifacts and enforcing the domain rules automatically makes DSLs useful not only for software developers, but for domain experts as well.

A textual DSL is *internal (or embedded)* if it is implemented by using a general-purpose language in a special, human-friendly way, for example, through macros or fluent interfaces [Fowler, 2010] [Kelly and Tolvanen, 2008]. This allows for the reuse of the existing general-purpose language tooling, which on the other hand, limits customizability and user-friendliness. Those textual DSLs that have their own syntax are called *external* DSLs. These require an own parser but do not impose any limitations on the syntax or on compiler messages.

DSLs are used for domain-specific problem descriptions. Because of the wide usage of DSLs, the artifacts created with them (the instances of the language) have several different names. When dealing with textual DSLs we usually speak about scripts, and visual DSL artifacts are generally called diagrams. We may also use the terms program and model for any of the two kinds of DSLs because DSL scripts and diagrams can be sometimes executable or in another case, it is practical to consider the stored information as a model instance.

2.4 Domain-Specific Modeling

The key concept behind model-based software methods is to express vital information in the model and let model processors accomplish the manual work of generating the code. This approach requires the model to use a representation comfortable to express vital information; furthermore, the model and the code generator together should provide all information required by the code generation. Domain-specific modeling and model processing can successfully address these requirements.

Domain-specific modeling-based software development fundamentally raises the level of abstraction while at the same time narrowing down the design space. With domain-specific languages, the problem is solved only once by modeling the solution using familiar domain concepts. A reasonable part of final products (source code, configuration files, and other artifacts) is then automatically generated from these high-level specifications with domain-specific code generators. The automation of application development is possible because the modeling language, the code generator, and the supporting framework need to fit the requirements of a narrow application domain.

To raise the level of abstraction in model-driven development, both the modeling language and the model processor (generator) need to be domain-specific. This approach improves both the performance of the development and the quality of the final products. The benefits of the method are improved productivity, better product quality, hiding complexity, and leveraging expertise.

We define a domain as an area of interest to a particular development effort. Domains can be a horizontal, i.e. technical, such as persistency, user interface, communication, or transactions, or vertical, i.e. functional, business domain, such as telecommunication, banking, robot control, insurance, or retail. In practice, each domain-specific modeling solution focuses on even smaller domains because the narrower focus enables better possibilities for automation and they are also easier to define.

Examining industrial cases and different application areas where models are used effectively as the primary development artifact, we recognize that the modeling languages applied were not general purpose but domain-specific. Some well-known examples are languages for database design and user interface development.

Using DSLs has the benefit that domain experts do not have to learn new (programming) languages. They can work with the already well-known domain concepts. By domain-specific modeling, domain experts define the business processes and domain requirements using the concepts of the domain.

Domain-specific models are rarely the final product of a modeling scenario. We can generate reports, document templates, or statistics from models. Moreover, the specialization makes it possible to develop a framework containing the base knowledge of the domain and generate code from models utilizing the features of this framework. As a result, we reduce the amount of error-prone manual mappings from domain concepts to design or to programming language concepts.

2.5 Semantics of Software Models

In computer science, the term semantics refers to the meaning of language constructs. Semantics provides the rules for interpreting the syntax which do not provide the meaning directly but constrains the possible interpretations of what is declared. Formal semantics, for instance, helps to write compilers, better understand what a program (model transformation) is doing, prove language statements, support optimization and refactoring by providing semantics and semantical comparison of models. There are many approaches to formal semantics; these belong to three major classes:

- *Operational semantics*. The meaning of a construct is specified by the computation it induces when it is executed on a machine. In particular, it is of interest how the effect of a computation is produced.
- *Denotational semantics*. The meaning is modelled by mathematical objects that represent the effect of executing the constructs. Therefore, only the effect is of interest, not how it is obtained.
- *Axiomatic semantics*. Specific properties of the effect of executing the constructs are expressed as assertions. Therefore, there may be aspects of the executions that are ignored.

Apart from the choice between denotational, operational, or axiomatic approaches, most variation in formal semantic systems arises from the choice of supporting mathematical formalism. Some variations of formal semantics include the following: action semantics, algebraic semantics, attribute grammars, categorical semantics using category theory as the core mathematical formalism, others.

2.6 Model-Driven Development and Model Processing

Developers generally differentiate between modeling and coding. Models are used for designing systems, understanding them better, specifying required functionality, and creating documentation. Code is then written to implement the designs. Debugging, testing, and maintenance are done on the code level as well. Quite often, these two different “media” are unnecessarily seen as being rather disconnected, although there are various ways to align code and models.

In model-driven development, we use models as the primary artifacts in the development process: we have source models instead of source code. It raises the level of abstraction and hides complexity.

Truly model-driven development uses automated transformations in a manner similar to the way a pure coding approach uses compilers. Once models are created, target code can be generated and then compiled or interpreted for execution. From a modeler’s perspective, generated code, of a specific area or component, is complete and it does not need to be modified after generation. This means, however, that the intelligence is not just in the models but also in the code generator and underlying framework. Otherwise, there would be no raise in the level of abstraction and we would be round-tripping again.

While making a design before starting implementation makes a lot of sense, most companies want more from the models than just throwaway specification or documentation that often does not reflect what is actually built.

Domain-specific modeling does not expect that all code can be generated from models, but anything that is modeled from the modelers’ perspective, generates complete finished code. Usually this is the

code of a software component or module. This completeness of the transformation has been the cornerstone of automation and raising abstraction in the past.

The generator is written by a company's own expert developer who has written several applications in that domain. The code is thus just like the best in-house code at that particular company rather than the one-size-fits-all code produced by a generator supplied by a modeling tool vendor.

Next section providing a classification discusses the popular model processing methods.

2.7 Classification of Model Transformation Approaches

There are several model transformation approaches ranging from relational specifications [Akehurst and Kent, 2002] to graph transformation techniques [Ehrig et al, 1999], to algorithmic techniques for implementing a model transformation. Based on [Czarnecki and Helsen, 2006] and [Mens and v. Gorp, 2006], we distinguish between the following approaches (Figure 2-1).

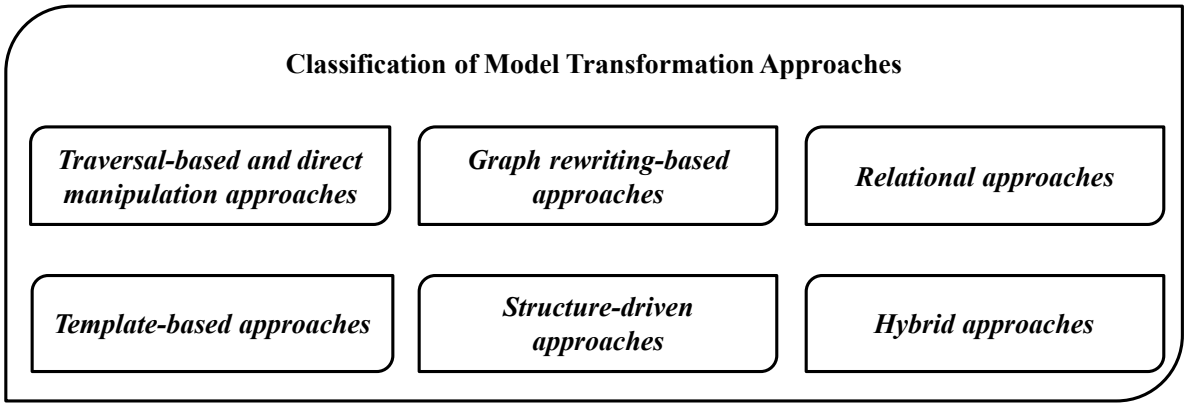


Figure 2-1 Classification of model transformation approaches

- *Traversal-based and direct manipulation approaches.* Traversing model processors provide mechanisms to visit the internal representation of a model and write text (source code or other text, e.g., XML) to a stream while optimizing and generating models and other artifacts. Furthermore, modeling and model processing approaches (aside from the model representation) offer some API to manipulate the models. These approaches are usually implemented using an imperative programming language [Vajk et al, 2009].
- *Template-based approaches.* Approaches in this category are mainly applied in the case of model-to-code generation. A template usually consists of the target text containing splices of source code (meta-code) used to access information from the source and to perform code selection and iterative expansion. The meta-code may be imperative program code or declarative queries as is the case with OCL [OMG OCL, 2014], XPath, or T4 Text Templates [Microsoft T4].
- *Relational approaches.* These approaches declaratively map between source and target models. This mapping is specified by constraints, which define the expected results, not the way in which they are achieved. Some examples of this are Query, Views, Transformations (QVT) [OMG QVT, 2016] and partially Triple Graph Grammars (TGGs) [Schürr, 1994].
- *Graph rewriting-based approaches.* Models are represented as typed, attributed, labeled graphs. The theory of graph transformation is used to transform models. Some examples of these approaches are AGG [AGG], AToM^3 [AToM3], GReAT [GReAT], TGGs, VIATRA2 [VIATRA2] and VMTS [VMTS].

- *Structure-driven approaches.* The transformation is performed in phases: the first phase is concerned with creating the hierarchical structure of the target model, whereas the second phase sets the attributes and references for the target (e.g. OptimalJ and QVT).
- *Hybrid approaches.* Hybrid approaches combine two or more of the previous categories (e.g., ATL [ATL] combines template-based, direct manipulation, and graph rewriting-based approaches. Another hybrid approach worth mentioning is TGGs).

One of the most popular model transformation approaches, taking both the literature and the industry into consideration, is the graph rewriting-based approach. In this method, the concentration is on the verification and validation capabilities of the graph transformation-based approaches. Therefore, the next section summarizes the theoretical foundations of this approach.

2.8 Graph Rewriting-Based Model Transformation

Graph rewriting-based transformations are a widely used technique for model transformation [Karsai et al, 2003] [de Lara et al, 2004]. Graph transformations have their roots in classical approaches to rewriting, such as Chomsky grammars and term rewriting [Rozenberg, 1997]. There are many other representations of this, which are not yet mentioned. In essence, a rewriting rule is composed of a left-hand side (LHS) pattern and a right-hand side (RHS) pattern.

Operationally, a graph transformation from a graph *G* to a graph *H* follows these main steps:

1. Choose a rewriting rule.
2. Find an occurrence of the LHS in *G* satisfying the application conditions of the rule.
3. Finally, replace the subgraph matched in *G* by the RHS.

There are many different graph transformation approaches applying the above steps [Rozenberg, 1997] [Syriani, 2009]. One of them is the popular algebraic approach, based on category theory with push-out constructs on the category, as seen in Graph [Ehrig et al, 2006]. Algebraic graph transformations have two branches: the Single-Push-Out (SPO) and the Double-Push-Out (DPO) approach.

The DPO approach has a large variety of graph types and other kinds of high-level structures, such as labeled graphs, typed graphs, hypergraphs, attributed graphs, Petri nets, and algebraic specifications. This extension from graphs to high-level structures was initiated in [Ehrig et al, 1991a] [Ehrig et al, 1991b] leading to the theory of high-level replacement (HLR) systems. In [Ehrig et al, 2004], the concept of high-level replacement systems was joined with adhesive categories, introduced by Lack and Sobocinski in [Lack and Sobocinski, 2004], leading to the algebraic construct of adhesive HLR categories and systems. In general, an adhesive HLR system is based on the double-push-out method. However, these are not only for the category of Graphs, also called rules, which describe, abstractly, how objects in this system can be transformed [Ehrig et al, 2006], Ehrig et al. provides a detailed presentation of adhesive HLR systems.

Graph transformations define the transformation of models. The LHS of a rule defines the pattern to be found in the host model; therefore, the LHS is considered the positive application condition (PAC). However, it is often necessary to specify what pattern should not be present. This is referred to as negative application condition (NAC) [Habel et al, 1996]. Besides NACs, some approaches [AGG] [VIATRA2] use other constraint languages, e.g., OCL, to define the execution conditions.

The scheduling of transformation rules can be achieved by explicit control structures or can be implicit due to the nature of their rule specifications. Moreover, several rules may be applicable at the same time. Blostein et al. [Blostein et al, 1996] have classified graph transformation organization in four categories.

(i) An *unordered* graph-rewriting system simply consists of a set of graph-rewriting rules. Applicable rules are selected non-deterministically until none are applicable. (ii) A *graph grammar* consists of the rules, a start graph and terminal states. Graph grammars are used for generating language elements and language recognition. (iii) In *ordered* graph-rewriting systems, a control mechanism explicitly orders the rule application of a set of rewriting rules (e.g., priority-based, layered/phased, or with an explicit control flow structure). (iv) In *event-driven* graph-rewriting systems, rule execution is triggered by external events. This approach has recently seen a rise in popularity [Guerra and de Lara, 2007].

Controlled (or programmed) graph transformations impose a control structure over the transformation rules to maintain a stricter ordering over the execution of a sequence of rules. Graph transformation, control structure primitives may provide the following properties: atomicity, sequencing, branching, looping, non-determinism, recursion, parallelism, backtracking and/or hierarchy [Lengyel, 2006] [Rozenberg, 1997].

Some examples of control structures are as follows: AGG [AGG] uses layered graph grammars. The layers fix the order in which rules are applied. The control mechanism of AToM^3 [AToM3] is a priority-based transformation flow. Fujaba [Fujaba] uses story diagrams to define model transformations. The control structure language of GReAT [GReAT] uses a dataflow diagram notation. GReAT also has a test rule construction; a test rule is a special expression that is used to change the control flow during execution. VIATRA2 [VIATRA2] applies abstract state machines (ASM). VMTS [VMTS] uses stereotyped UML activity diagrams to further specify control flow structures. The model transformation process is depicted in Figure 2-2. In [Taentzer et al, 2005], a comparative study is provided that examines the control structure capabilities of the tools AGG, AToM^3, VIATRA2, and VMTS.

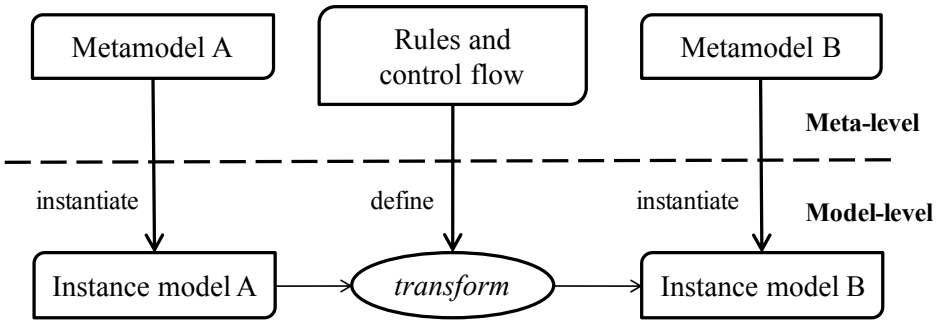


Figure 2-2 Overview of the graph rewriting-based model transformation process

2.9 A Modeling and Model Transformation Framework

More than fourteen years ago, our research team has analyzed existing modeling frameworks. We have found that it is possible to create a solution, which is highly customizable, but fast and efficient as well. We have created our own modeling and model-processing framework, Visual Modeling and Transformation System [VMTS]. Since then, we have fine-tuned the framework several times based on the industrial requests and the experiences gained. Current version of VMTS is heavily based on generative techniques [Czarnecki and Eisenecker, 2000] and uses a modular structure. Generative techniques are used to create efficient and highly flexible APIs from domain definitions, while the modular design helps in creating a wide range of applications based on these APIs. The result is a framework, where the user can decide at *run-time* whether to use customizability features or performance optimized version, furthermore, we can also choose the appropriate storage type (e.g. file, database, cloud storage). VMTS also offers customizable graphical and textual editors for editing the domain models.

The Visual Modeling and Transformation System is a graph-based, domain-specific (meta)modeling and model processing framework, or more likely a complete domain modeling platform. VMTS addresses the diversity, various storage types, model processors and performance issues of the modeling area. During its design and improvement, we have continuously taken into consideration the requests from our industrial partners, cutting-edge technology solutions, knowledge of other modeling frameworks, and the conclusions of workshops, where VMTS has been presented.

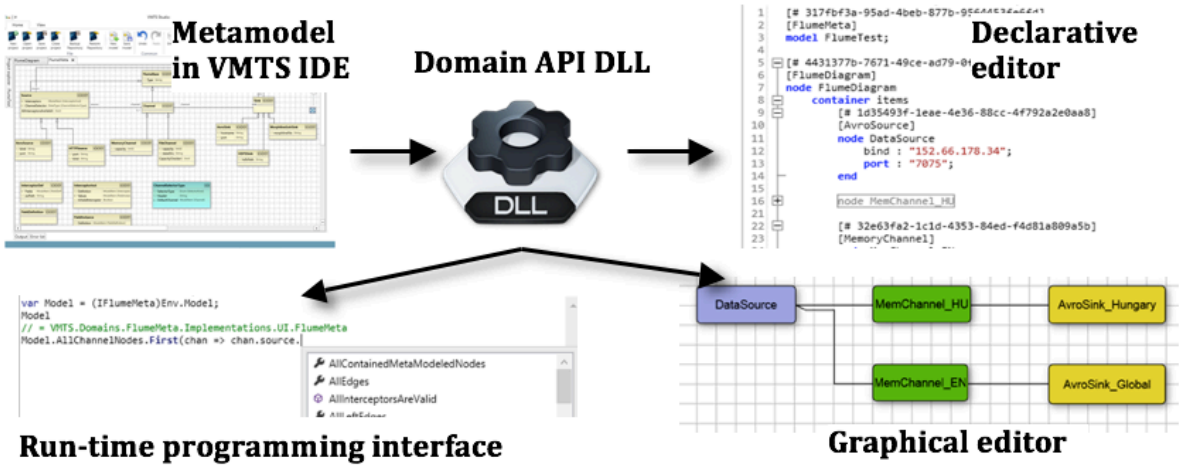


Figure 2-3 The VMTS domain modeling platform

In VMTS, we create domain definitions in a modeling IDE referred to as VMTS Studio. Domain definitions are described by a graphical language similar to UML Class diagram. Domain entities, relations between entities, their attributes and operations can be specified in a user-friendly way. From the domain definition, we can generate an assembly (a .dll file), which represents a domain-specific API. The domain assembly allows us to load, or save domain models (in several formats and media types) and acts as a base for all VMTS-related applications. For example, we can build or generate a graphical editor for domain models, we can edit models as a declarative document or we can also use the built-in run-time script evaluator and execute changes on models – using the domain API – on-the-fly. Although the IDE of VMTS is highly customizable for domains, we can also build a lightweight, standalone application not using the IDE, but still working on the domain models, enforcing the domain rules and supporting model storage functionality.

VMTS is used not only for modeling, but for model processing (code generation) as well. The first step here is always to ensure that the model is valid. Theoretically, the domain definition does not allow invalid models at all; however, code generators may also have certain additional requirements. By using the domain API, these validations can be appied. Model processing could be relatively simple, since we have to traverse the model objects and add the appropriate code fragments to the result on each model element.

The majority of the results presented in the thesis are implemented, measured and validated with the help of the VMTS framework.

3 Methods for Verifying and Validating Graph Rewriting-Based Model Transformations

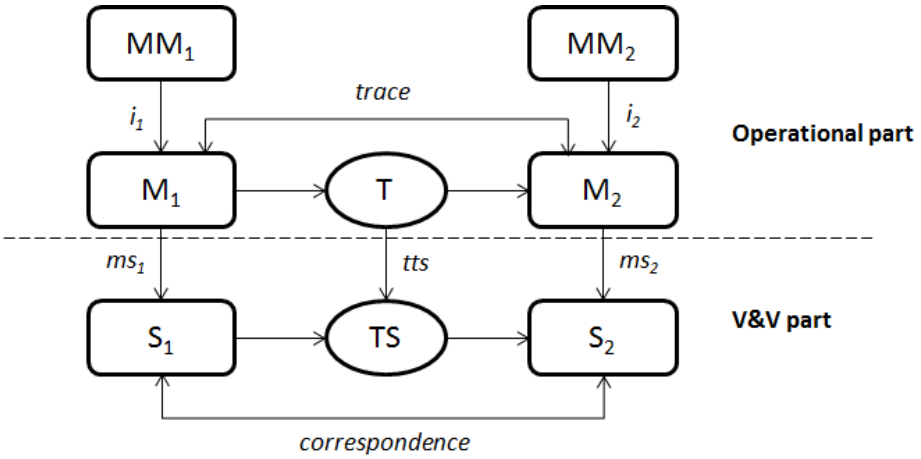
3.1 Introduction

Both the complexity of software systems and the necessity for reliable systems increases. This results that the verification and validation of model transformation-based approaches and the specification of model transformations become emerging research fields. This section of the thesis provides a classification framework for model transformation verification and validation approaches. We introduce three views, *Path View, Computational View,* and *Property View,* which serve as the pillars of our classification system.

The *Path View* allows the classification of model transformation verification and validation approaches. The *Computational View* analyses the complexity aspect of verification and validation approaches. The *Property View* allows to explore the properties of both the model transformation definition and the target models which can be verified and/or validated by model transformation approaches [7] [10].

The classification framework assists (i) end-users and the model transformation communities' awareness of property classes that can and should be verified/validated by model transformations, (ii) software developers to choose a particular model transformation approach that is best suited for their needs, (iii) tool builders to provide the strengths and weaknesses of certain tools when compared to others. Moreover, it can (iv) assist scientists in identifying approach-related limitations that need to be overcome by improving the underlying techniques and formalisms.

We address the following questions: "What type of model and model transformation properties can be verified/validated? What are the existing model transformation approaches and tools that support verification and/or validation? Is it possible to develop automated validation and/or verification techniques for graph transformation systems? What are the model transformation verification and/or validation-related open issues?"



In Figure 3-1 The *Paths* of model transformations

First, we discuss the different scenarios of model transformation verification and validation. We refer to these scenarios as *Paths*. In Figure 3-1 depicts the paths: the top half of the figure represents the *Operational part*, and the bottom half depicts the *verification/validation (V&V)* part. The *Operational part* is designed by the transformation engineer.

The related verification/validation questions are as follows: Can we verify a property in one of the operational domains (e.g. in the source model M_1, the transformation T, or in the target model M_2)? If not, what other domains need to be involved (which *Path* of the In Figure 3-1 should be taken), where the verification/validation can be performed or more aptly formed? In what way is the mapping arranged between the operational domains and the verification/validation domains?

Sometimes properties that will be verified/validated cannot be expressed in the operational domains. To address this, we have introduced the *verification and validation* (*V&V*) part including additional domains. In In Figure 3-1, MM_1 and MM_2 are the language specifications (metamodels) and can define only two domains. In special cases, MM_1 and MM_2 can be identical. M_1 and M_2 are instance models of MM_1 and MM_2 respectively. The instantiation is defined by the mappings i_1 and i_2. Transformation T converts M_1 into M_2. The mapping *trace* stores the relation between elements of models M_1 and M_2. Based on this mapping, for each element of model M_1, we can identify the appropriate target model elements (image) in model M_2, and vice versa; for each element of model M_2, we can identify the appropriate source model elements in model M_1. Our goal is to verify the semantic correctness of the transformation T; therefore, if the formalism used by M_1, M_2 and T is not adequate, then they are mapped into a different semantic domain. Their images are S_1, S_2, and TS, respectively. The mapping is defined by ms_1, ms_2, and tts. The correspondence between S_1 and S_2 is a specific knowledge: a special semantic relationship expected by the transformation designer. This *correspondence* is verified in a semantic domain. Next, assuming that the mappings, (ms_1, ms_2 and tts) from the domain-specific artifacts (M_1, M_2 and T) into the semantic domain (S_1, S_2, and TS), are correct, we can reason the correctness of the transformation T [1] [7] [10].

Recall in In Figure 3-1, we demonstrated a general case scheme, incorporating several special cases. The introduced scheme represents a one-way transformation, but the bi-directional scenario can be constructed by repeating this structure in the opposite direction. In a general case, the two directions require different mappings; only in special cases can the same mapping be applied.

Based on the architecture of the *Paths*, we have identified the following semantic verification/validation types:

1. Verification of the models M_1 and M_2 ($Path^{Models}$).
2. Verification of the transformation T ($Path^{TransT}$).
3. Verification of the transformation TS ($Path^{TransTS}$).
4. Verification of the correspondence ($Path^{Corresp}$).
5. Hybrid verification: combines two or more of the previous verification types ($Path^{Hybrid}$).

Each of these verification/validation types defines a path. During the verification and validation, we traverse the paths of the framework in the following ways.

$Path^{Models}$. Verification of the models M_1 and M_2 means both M_1 and T (M_1) are verified separately. This type of verification does not attempt to prove the validity of the graph transformation T, but verifies that both of the models provide an appropriate solution to the problem. Typically, the conformance into metamodels is validated with this path: the modeling tool allows the creation of appropriate model elements only, while a constraint checker (e.g., OCL checker) is executed on the source model (M_1). Next, the transformation T processes the model and generates the output model (M_2) which conforms to the output metamodel (MM_2). The validation of the output model M_2 is performed again by the modeling environment: validation of the metamodel convergence, including constraint checking.

$Path^{TransT}$. Most of the verification/validation approaches aim to check the correctness of the transformation rules in general. There are both static (offline) and dynamic (online) approaches as well. For example, Asztalos et al. [22] developed a formal language that is able to express a set of model transformation properties. Basically, the language is appropriate to specify both the properties of the output models and the properties of the relation between the input and output model pairs. They introduced a final formula that describes the properties that remain true at the end of the transformation. The approach is able to derive the proof or refutation of a verifiable property from the final formula. An example dynamic approach is [Lengyel, 2006], in which the validation of the transformation is achieved with constraints assigned to the transformation rules as pre- and postconditions.

$Path^{TransTS}$. In the most generic case, transforming models into other domains means a projection from the source language to the target language, possibly with an intentional loss of information. Therefore, in certain cases, proving full semantic equivalence between source and target models is not the objective. Instead, we can define transformation or language-specific (source and target domain) validation properties that should be satisfied by the transformation.

The transformation definition describes the required model manipulation either in an imperative or a declarative (mostly relational) way. This representation is often inappropriate as a subject of verifying certain properties. Therefore, we map the transformation to a domain more suitable to perform formal verification/validation. There are several approaches that map M_1, M_2 and T into a semantic domain and perform the verification either on the image of the transformation (*TS*: $Path^{TransTS}$) or on the *correspondence* between the images of the source and generated models (S_1 and S_2: $Path^{Corresp}$).

An example of $Path^{TransTS}$ is provided in [Varró et al, 2006]: model transformations are mapped into Petri nets with the goal of performing the termination analysis in a more appropriate domain.

$Path^{Corresp}$. The correspondence relation between the S_1 and S_2 is domain-specific knowledge; this is the semantics expected by the language and/or model transformation designer. We should realize that the source and the target domains (MM_1 and MM_2) could be quite distant from each other (e.g., abstraction level, domain concepts, or model structure). Thus, the correspondence may be an optional domain-specific knowledge that represents the semantic mapping between the images of source and target models in a semantic domain.

In the context of our verification/validation classification framework, the equilibrium (property preservation) between the source and generated models, that most of the approaches attempt to verify (e.g., [Giese et al, 2006], [de Lara and Taentzer, 2004] and [Varró et al, 2003]), is a special mapping among the source and the target domains. Similarly, other special mappings have already been configured, e.g., bi-similarity: two systems can be said to be bi-similar if they behave in the same way, i.e., one system simulates the other, and vice versa [Narayanan and Karsai, 2008].

An example for $Path^{Corresp}$ is the following: within the domains of the source and the target modeling languages, it is hard to prove the correctness of the design. Therefore, the models are projected into a formal domain, such as transition systems, and the formal analysis is performed in this domain e.g., by applying bi-simulation [Narayanan and Karsai, 2008].

$Path^{Hybrid}$. This path combines two or more of the paths introduced above. For example, a certain development scenario requires to verify/validate both transformation termination and some domain-specific properties. $Path^{TransTS}$ is applied to verify termination and $Path^{TransT}$ is applied to validate the required domain-specific properties, e.g., attribute value requirements.

3.2 The Dynamic Validation Method

3.2.1 An Example

Figure 3-2 depicts the metamodel of a domain-specific language. This language defines that an instance model contains *Domain* objects. A domain can contain sub-domains and domains can also be linked to each other. A domain has *Server* objects. A server must belong to a domain. A server contains a *ServerName*, *Id*, *Type* (enum attribute with values *Web*, *Database*, *Mail*, and *Gateway*), and *Load* attributes. *Servers* contain sequentially ordered *Tiers*. A tier has the following attributes: *TierName*, *Id*, *Type* (enum attribute with values *CPU* and *I/O*), *OrderId*, *ServiceTime*, and *VisitNumber*. Each server has exactly one *ThreadPool* element. A *ThreadPool* is comprised of *ThreadPoolName*, *Id*, and *MaxNumberOfThreads* attributes. The *ThreadPool* contains *Threads*. Each thread has *Id* and *State* (enum attribute with values *Ready* and *Occupied*) attributes. Servers have one or more *Queues*. A queue has *QueueName*, *Id*, and *QueueLimit* attributes. A queue must belong to a server. Furthermore, servers and queues can contain *Tasks*. *Tasks* assigned to servers are under processing, while tasks in a queue are in waiting state. A task has the following attributes: *TaskName*, *Id*, *Priority* (enum attribute with values *Normal*, *High*, and *Urgent*), and *ProcessedState* (enum attribute with values *Waiting*, *Processing*, and *Complete*). There are also more specific task types inherited from *Task*: *Email*, *BulkEmail*, *WebRequest*, *DBRequest*, and *AuthorizationRequest*. Each of these metamodel elements also includes further attributes.

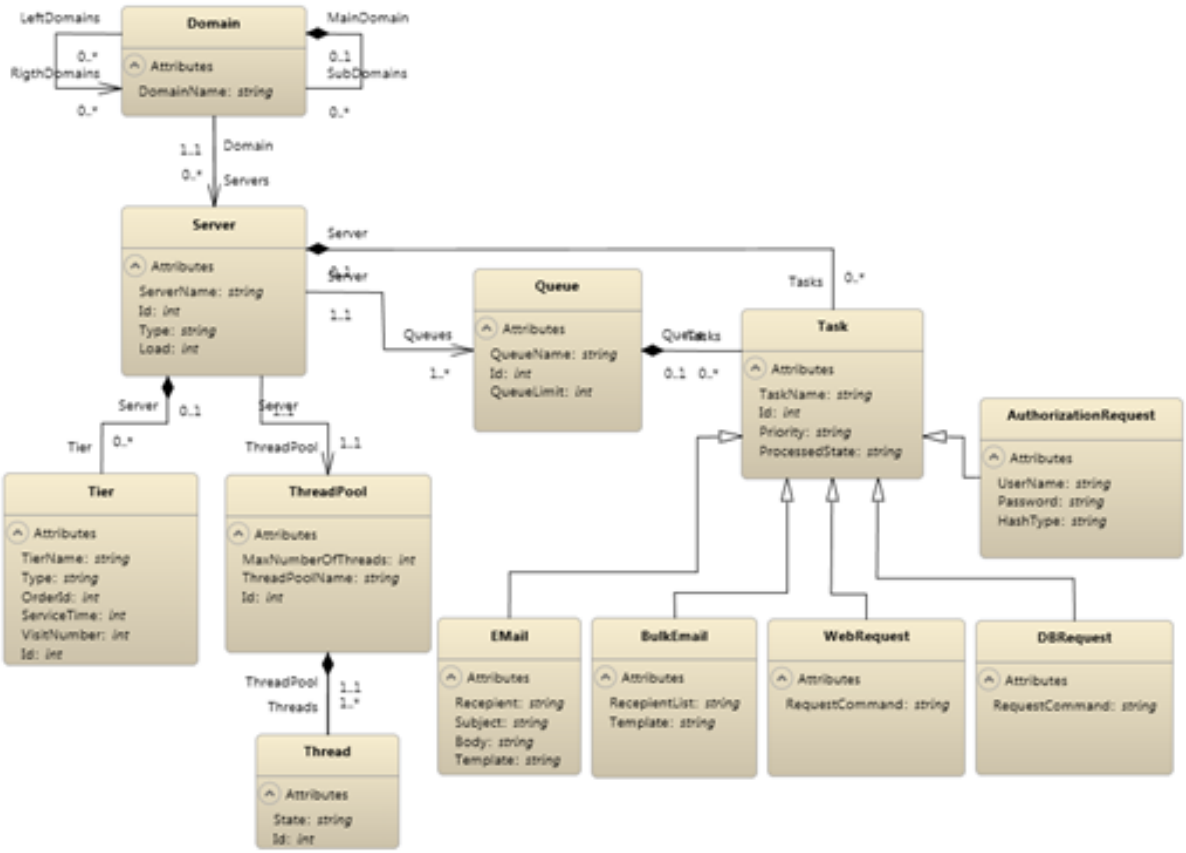


Figure 3-2 The *DomainServers* metamodel

Figure 3-3 introduces a control flow model of a rule-based system. The processing has three transformation rules. The rule *CheckServerLoad* selects a *Server* which *Load* is over 80%. If there is no such server, then the transformation terminates. Otherwise, a new server node, with a *ThreadPool* and

a *Queue* node, is inserted into the domain. Next, the transformation rule, *RearrangeTasks*, rearranges tasks from the queue of the overloaded server to the queue of the new server. The *RearrangeTasks* rule is executed in *Exhaustive* mode: it is continuously applied while the *Load* of the overloaded servers is over 70%, and the *Load* of the new server remains under 70%. The transformation is executed in a loop. This means, after easing the load of one server, the process continues and therefore, the transformation can insert additional, new servers.

Figure 3-3 Example model transformation: *LoadBalancing*

Figure 3-4 depicts two example rules: *AddNewServer* and *RearrangeTasks*. The figure follows a compact notation, containing no separated LHS and RHS pattern. The colors code the following: black nodes and edges denote unmodified elements, blue ones indicate newly created elements, and red ones mark the elements deleted by the rule. The transformation rule *AddNewServer* gets the *Domain* type node as a parameter and creates the new *Server* with a *ThreadPool*, two *Threads*, a *Tier*, and a *Queue*. The transformation rule, *RearrangeTasks*, receives the two servers with their queues as parameters and performs the rearrangement as a single task. The rule is executed in *Exhaustive* mode, which enables several tasks to be moved between the queues.

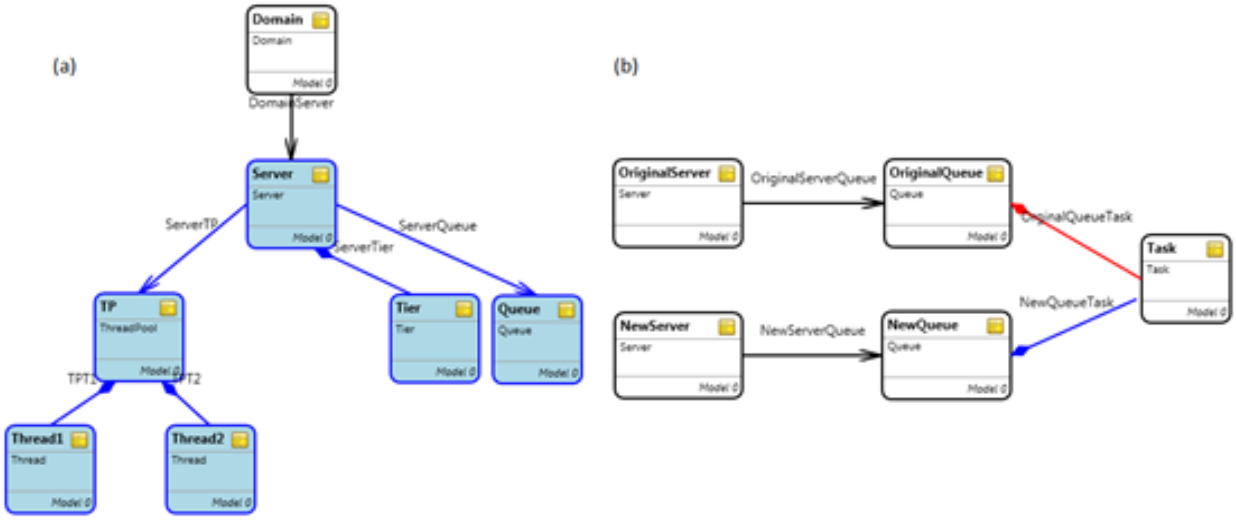


Figure 3-4 Example model transformation rules: (a) *AddNewServer* and (b) *RearrangeTasks*

Some example constraints assigned to the rules are as follows:

```
context Server inv serverCardinality:
Server.allInstances()->count() < 40

context Queue inv queueCardinality:
Queue.allInstances()->count() >= Server.allInstances()->count()
```

The constraints *serverCardinality* and *queueCardinality* define the number of specific type elements in the model. These are cardinality issues related to the whole model.

```
context Queue inv queueLimit:
QueueLimit < 1500
```

The constraint *queueLimit* is an attribute value constraint that maximizes *QueueLimit* attribute of *Queue* type nodes.

```
context Server inv largeThreadPools:
Server.allInstances->forall(s | s.ThreadPool.Threads->count()<= 50 OR
      (s.Tiers->exists(t | t.Type = Type::CPU) AND
       s.Tiers->exists(t | t.Type = Type::I/O)))
```

The constraint *largeThreadPools* defines that for each server, if the number of threads in the *ThreadPool* exceeds 50, then separated CPU and I/O tiers are employed.

The presented constraints are assigned to the rules and guarantee our requirements. After a successful rule execution, the conditions hold and the output is valid. The fact that the successful execution of the rule guarantees the valid output cannot be achieved without these validation constraints.

3.2.2 A Validation Method for Rule-Based Systems

A rule-based system [Hayes-Roth, 1985] [Williams and Bainbridge, 1988] is a series of *if-then* statements that utilizes a set of assertions, to which rules are created on how to act upon those assertions. Rule-based systems often construct the basis of software artifacts, which can provide answers to problems in place of human experts. Such systems are also referred as expert systems. Rule-based solutions are also widely applied in artificial intelligence-based systems, and graph rewriting is one of the most frequently applied implementation techniques for their realization. As the necessity for reliable rule-based systems increases, so emerges the field of research regarding verification and validation of graph rewriting-based approaches. In this section, we provide a dynamic (online) method to support the validation of algorithms designed and executed in rule-based systems. The proposed approach is based on a graph rewriting-based solution.

Rule-based systems provide an adaptable method, suitable for a number of different problems. Rule-based systems are appropriate for fields, where the problem area can be written in the form of *if-then* rule statements and for which the problem area is not extremely too expansive. In case of too many rules, the system may become difficult to maintain and can result in decreased performance speeds.

A classic example of a rule-based system is a domain-specific expert system that uses rules to make deductions or narrow down choices. For example, an expert system might help a doctor choose the correct diagnosis based on a dozen symptoms, or select tactical moves when playing a game. Rule-based systems can be used in natural language processing or to perform lexical analysis to compile or interpret computer programs. Rule-based programming attempts to derive execution instructions from a starting set of data and rules. This is a more indirect method than that employed by an imperative programming language, which lists execution steps sequentially.

As rule-based systems are being applied to many diverse scenarios, there is a need for methods that support the verification and validation of the algorithms performed by these systems. Verification and validation of a rule-based system is the process of ensuring that the rules meet specifications and fulfill their intended purpose.

We have reviewed the different control structures of model transformations in Section 2.7. In the case of rule-based systems, the application order of the rules is supported by a conflict resolution strategy. The strategy may be determined by the actual area or may simply be a matter of preference. In any case, it is vital as it controls which of the applicable rules are fired and thus the behavior of the entire system. The most common strategies are as follows:

- *First applicable:* If the rules are in a specified order, firing the first applicable rule allows for control over the order in which rules are fire.
- *Random:* Though it does not provide the predictability or control of the first-applicable strategy, it does have certain advantages. For one, its unpredictability is an advantage in some circumstances (e.g., in games). A random strategy simply chooses a single random rule to fire from the conflict set. Another possibility for a random strategy is a fuzzy rule-based system in which each rule has a factored probability, i.e., some rules are more likely to fire than others.
- *Least recently used:* Each of the rules is accompanied by a time or step stamp, which marks the time of its last usage. This maximizes the number of individual rules that are fired at least once. This strategy is perfect when all rules are needed for the solution of a given problem.
- *Best rule:* Each rule is given a weight, which specifies its comparative consideration to the alternatives. The rule with the most preferable outcomes is chosen based on this weight.

Transformation rules can be made more relevant to software engineering models if the transformation specifications allow the assigning of validation constraints to the transformation rules. The objective of this research activity is to support the verification and validation of algorithms performed by rule-based systems. The requirements, assigned to the rules are both input and output related requirements, i.e. we define certain pre- and postconditions that should hold before and after the execution of the rule. In several cases, rules do not contain certain node or edge types that are about to be included into our verification and validation requirements. These requirements may relate to a temporary (during the processing) or a final (following the processing) state of the input or generated models. Moreover, several different directions can be followed; e.g. we can assert additional requirements to the input and output models (metamodel constraints), or the rule-based system can be extended with the use of appropriate testing and validating rules.

Dynamic validation covers both the attribute value and the structure validation, which can be expressed in first-order logic extended with traversing capabilities. Example languages currently applied for defining attribute value and interval conditions are Object Constraint Language (OCL), C, Java, and Python. These conditions and requirements are pre- and postconditions of a transformation rule.

Definition (Precondition). A precondition assigned to a rule is a Boolean expression that must be true at the moment of rule firing.

Definition (Postcondition). A postcondition assigned to a rule is a Boolean expression that must be true after the completion of a rule.

If a precondition of a rule is not true, then the rule fails without being fired. If a postcondition of a rule is not true after the execution of the rule, the rule fails.

With pre- and postconditions the execution of a rule is as follows (Figure 2-2):

1. Finding the match according to the LHS structure.
2. Validating the constraints defined in LHS on the matched parts of the input model.
3. If a match satisfies all the constraints (preconditions), then executing the rule, otherwise the rule fails.
4. Validating the constraints defined in RHS on the modified/generated model. If the result of the rule satisfies the postconditions, then the rule was successful, otherwise the rule fails.

A direct corollary is that an expression in LHS is a precondition to the rule, and an expression in RHS is a postcondition to the rule. A rule can be executed if and only if all conditions enlisted in LHS are true. Also, if a rule finished successfully, then all conditions enlisted in RHS must be true.

If a finite sequence of rules is specified properly with the help of validation constraints, and the sequence of rules has been executed successfully for the input model, then the modified/generated output model is in accordance with the expected result that is described by the finite sequence of transformation rules refined with the constraints.

Definition (Low-level construct). Pre- and postconditions defined as constraints and propagated to the rules are low-level constructs.

Definition (High-level construct). Validation, preservation and guarantee properties are high-level constructs.

Definition (Validated rule execution). A rule execution is validated if it satisfies a set of high-level constructs.

To summarize, high-level constructs define the requirements in a higher abstraction level, e.g. *only non-abstract classes should be processed*. Low-level constructs are the appropriate constraints assigned to the appropriate rules. These constraints facilitate to achieve the required conditions.

This method can be followed in Figure 3-4. Finding the structural match, the preconditions are validated, and after performing the rule execution, postconditions are validated. Both of the validations should be successful in order to the whole rule be successful.

With this method, the required properties can be defined on low-level, i.e. on the level of the rules. In summary, we can state that the presented dynamic approach guarantees that if the execution of a rule finishes successfully, the generated output is valid and fulfills the required conditions. The validation of a rule-based system can be achieved with constraints assigned to the rules as pre- and postconditions [27] [36].

3.3 Model Transformation Property Classes

This section investigates the following question: what can be and what needs to be verified/validated by model transformations? In order to provide an answer for the question, we introduce the verification and validation related property classes. The proposed property classes identify the different verification and validation categories and discusses their specialties. The property classes make possible to clearly define verification/validation requirements stated against model processors, furthermore, they help to classify the existing approaches according to the verification and validation questions.

Based on [Abramski et al, 1993] and [Bundy, 1986], we can distinguish three main approaches: the mathematical approach (graph transformation systems based on set theoretic or algebraic

characterizations), the formal approach (logical characterizations-based graph transformation systems), and the model checking approach (translating graph transformation systems to Kripke-style models and performing proofs via model checking).

The mathematical approach is expressive, domain-specific, intuitive, yet difficult to apply in an automated way. The formal approach is also expressive and it is easier to apply in a programmed way. The model checking approach is less expressive, but offers the most with respect to automation [Bisztray et al, 2008b].

In the case of domain-specific modeling languages (DSMLs), the properties are domain-specific as well, meaning they require versatile transformation properties to be verified. The emerging questions are as follows: What are the properties that have been verified so far by researchers in the field? What are the model transformation properties provided by a particular language or approach? What further domain properties can be preserved or guaranteed by a transformation process / transformation language or approach? In order to answer these questions and categorize both the transformation and target domain-related properties, we introduce the property classes.

A property class is a set of properties regarding model transformations or models. A verification approach supports a property class if the model transformation is able to reason the properties belonging to the property class.

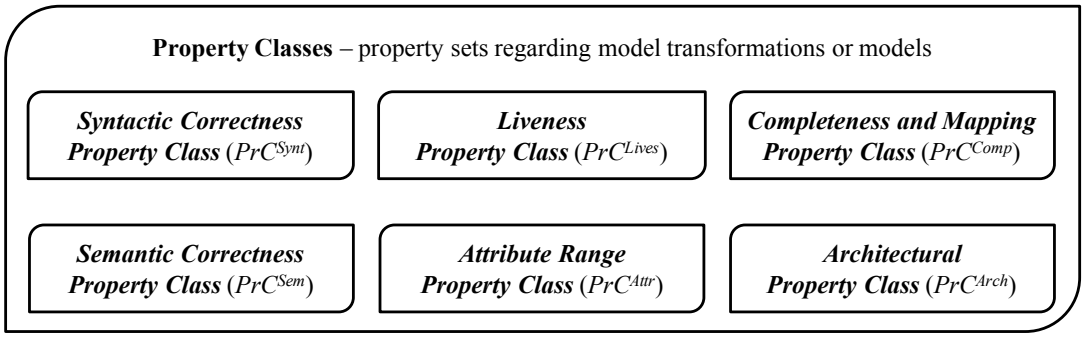


Figure 3-5 Property classes

Our property classes (Figure 3-5) cover both the properties that the already existing methods are able to verify/validate, and the ones for which verification and validation would be useful. We take different types of properties into account: (i) properties independent from modeling languages (source and target domain), e.g., termination and global determinism, and (ii) properties specific to transformation or modeling languages, e.g., semantic properties. The following list defines our property classes. These property classes have a significant effect on the usability of model transformations and on the quality of both model transformations and produced artifacts.

- ***Syntactic Correctness Property Class (PrC^{Synt}).*** The simplest notion of correctness is syntactic correctness: given a well-formed source model, it should be guaranteed that the produced target model is also well-formulated. Model processing environments handle this requirement in two different ways: in the first case, they do not require the syntactic correctness of the output model. This responsibility is propagated to the compilers of the target platform: compilers validate the syntactic correctness during the compilation process. In the second case, model processing environments require the syntactic correctness of the output model. For example, there are tools and approaches (e.g., [Anastasakis et al, 2007] [Cabot et al, 2008]) that can be used at the end of model transformations to validate the generated output. Other model transformation tools and

approaches (e.g., the BOTL approach [Braun and Marschall, 2003]) address the problems in ensuring that a model transformation always produces syntactically correct models.

- ***Liveness Property Class (PrC^{Lives}).*** This property class includes control flow-related properties, namely, the termination and global determinism. Termination and global determinism mean that, given a set of transformation rules, their application should always lead to a result, i.e., they should terminate in a finite number of steps, and their results should be unique. These are general and modeling language-independent semantic criteria for model transformations.
- ***Completeness and Mapping Property Class (PrC^{Comp}).*** The property class includes the completeness and three mapping properties. Syntactic completeness requires that, for each element in the source model, there should be a corresponding element in the target model that can be created by the model transformation. In general, a transformation system is complete if, for each valid source model, there is a valid target model satisfying the source model. Syntactic completeness was discussed in [Varró et al, 2003] by planner algorithms, and in [Hausmann et al, 2002] by graph transformation. The three mapping properties are as follows [Cabot et al, 2010]:
  - *Injective*: a transformation system is injective if each target model has exactly one single source model.
  - *Surjective*: a transformation system is surjective if each target model can be produced from a source model. This property is the reciprocal of property completeness.
  - *Bijective*: a transformation system is bijective if it is injective and also surjective.
- ***Semantic Correctness Property Class (PrC^{Sem}).*** With this property class, we require the fulfillment of certain semantic rules. The property class covers the domain-specific knowledge included in the semantics expected from the transformation. The goal is that the produced target model contains the expected semantic properties.

  The domain-specific semantic properties could be verified on the transformation *T*, on its image (*TS*), or on the domain-specific correspondence between the images of the source and the generated models in a semantic domain. Typically, semantic properties are verified either on the transformation itself (*T*) or with the analysis of the domain-specific correspondence.
- ***Attribute Range Property Class (PrC^{Attr}).*** This property class covers the attribute value and interval validation. An example is the definition of attribute constraints with Object Constraint Language (OCL) [OMG OCL, 2014]. The validation has a local nature and the properties are expressible in first-order logic, extended with traversing capabilities.
- ***Architectural Property Class (PrC^{Arch}).*** Architectural properties are output model related properties. These properties are invariants, reachability and topological properties. They are often expressible in second-order logic. Some notions can be modeled in modal logic and verified by model checkers, others may require more expressive languages.

The following subsections provide examples and detailed discussions related to these property classes and the incorporated properties. In each section, some important questions are investigated with respect to verified/validated model transformation and a number of different approaches, that provide concrete answers to our questions, are introduced. Based on our property classes, we are approaching answers to our motivating questions.

3.3.1 Syntactic Correctness Property Class – PrC^{Synt}

Referring to the transformation result, the minimal requirement is to assure syntactic correctness, i.e., to guarantee that the output model is a syntactically well-formulated instance of the target language. However, often syntactic correctness properties are motivated by semantic notions, e.g., in an automotive domain, an engine can belong to only one car. This requirement is semantically motivated by the domain, but expressed with a multiplicity condition. In a stricter sense, we can state that there exist cases in which the syntactic correctness property class verifies semantic properties.

As was mentioned earlier, model processing environments either do not ensure the syntactic correctness of the output model, or they provide additional features to validate the syntactic correctness of the output model.

In the first case, the syntactic validation of the output model is propagated to a separate tool or to the target platform compilers. There are several tools and approaches addressing the consistency or satisfiability problem for UML/OCL models [Anastasakis et al, 2007] [Cabot et al, 2008]: given a UML class diagram annotated with OCL constraints it is necessary to decide whether there exists a legal instance of the model which satisfies all graphical and OCL constraints. These tools and approaches can be used at the end of model transformations to automatically validate the generated output. The same is true of the current modeling environments that provide further domain-specific language (DSL) support beyond the standard UML, i.e. designing and using DSLs [Kelly and Tolvanen, 2008].

In the second case, the syntactic validation of the output model requires additional effort at the end of the model processing. The modeling environment enforces the creation of syntactically correct input models. Next, a model processing tool generates the output model that should also be checked in accordance with the output language definition. Currently, besides the syntactic correctness validation of the model transformation definition, most of the graph rewriting-based model transformation tools provide certain validation possibilities for syntactic correctness of the models produced by model transformations.

Syntactic Correctness of a Transformation Rule. A model transformation is formed by transformation rules. Within a model transformation, it is required that each rule be defined using an attributed type graph. This type graph is obtained by regarding the metamodels of both the source and the target languages. As a consequence, one requirement for rules is that both LHS and RHS be instances of their corresponding metamodels.

Often this instance relationship is applied in a loose sense: we require that a model must be an instance of the metamodel but should not necessarily conform to additional constraints (e.g. OCL constraints) [Küster, 2006]. It should be noted that it does not affect cardinality constraints. There are two reasons for this approach: (i) we do not want to define unnecessary restrictions when designing a model transformation i.e., always requiring a complete model as LHS or RHS. (ii) We want to reduce the number of rules by allowing abstract classes. Furthermore, certain tools (e.g., GReAT and VMTS) allow us to use temporary edges and nodes during the transformation. These temporary edges and nodes are created by the transformation, utilized during the execution, and deleted by the transformation in order to be free of temporary elements at the end of the model processing.

Syntactic Correctness of a Transformation. The control expression of a transformation also has a metamodel that defines the control structures. The control expression must be a valid instance of its metamodel and can only reference accessible transformation rules, i.e., transformation rules defined by the same transformation.

Küster [Küster, 2006] concentrated on checking syntactic correctness in which the following two types can be distinguished. When a language for expressing model transformations is available, then a concrete model transformation must be syntactically correct, with respect to this language. It is worth mentioning that such a model transformation language is not always mathematically formalized, but the execution semantics may be hidden in a model transformation tool. A rather different form of syntactic correctness is obtained by looking at the result of a model transformation. In this case, it is always desirable that the results conform to the syntax of some target language.

3.3.2 Liveness Property Class – PrC^{Lives}

The *liveness* property class incorporates the termination and the global determinism properties of model transformations. This property class ensures that, given a source model, a model transformation always produces a unique target model as a result. In the case of a non-terminating model transformation, the transformation is not permitted to terminate on certain models. In the case of non-determinism, the transformation may produce different target models when being applied to the same source model.

3.3.2.1 Termination of Model Transformations

A transformation terminates if it is ensured that the transformation stops after a finite number of steps. In general, the termination of a graph rewriting system is indeterminate [Plump, 1998]. Therefore, in general, given a graph transformation system, there are no mechanisms that support the automated decision whether or not to terminate the system. Different termination criteria have been presented in several papers [Bottoni et al, 2000] [Ehrig et al, 2005] [Levendovszky et al, 2007] and can be checked to prove the termination. These approaches are based on layered grammars or measurement functions of the execution order of rules. Therefore, termination can be achieved by providing a set of sufficient criteria.

Levendovszky et al. [Levendovszky et al, 2007] provided termination criteria for graph and model transformation systems with injective matches and a finite input structure. The approach proposes a treatment for infinite sequencing of rule applications and accounts for attribute conditions, negative application conditions, and type constraints. The termination criteria introduced by this approach provides general productions allowing recursion within the scope of DPO and typed attributed graph transformation. This is the theoretical basis to proving that certain control flows of rules are terminating.

Assmann [Assmann, 2000] introduced termination criteria for graph rewriting applied to program transformation. The approach assumes that there cannot be parallel edges, with the same labels, between two nodes. This leads to a termination criteria for specific rules if the label and node sets are finite. The approach introduces the subtractive rule, which is conceptually similar to deletion layers examined by Ehrig et al. [Ehrig et al, 2005].

Bottoni et al. [Bottoni et al, 2000] developed a theory for the DPO approach. It provides abstract termination criteria via a measure function. Concrete termination criteria, based on the number of nodes and edges, are provided.

Ehrig et al. [Ehrig et al, 2005] provided significant results regarding layered grammars. These results correspond to and build upon the contributions provided by de Lara and Taentzer [de Lara and Taentzer, 2004] and Bottoni et al. [Bottoni et al, 2000]. The criterion provided guarantees that the creation of all objects of a type should precede the deletion of the object of the same type. Thus, a layer deletion of an object of a certain type cannot create such objects, neither can the subsequent rules. This means that the productions in a deletion layer terminate for the reasons detailed above if object types are taken into consideration.

In general, concerning the termination of a transformation unit with control, the following observations can be made: each rule that is applied only once does not pose a problem because it cannot lead to infinite rule application sequences. However, if a set of rules can be applied as long as possible, it must be guaranteed that there are no occurrences of infinite rule application.

3.3.2.2 Global Determinism of Model Transformations

The termination of a transformation unit can be expressed through the analysis of a set of rules, being applied as long as possible until their termination. However, the global determinism of a transformation unit is more difficult to prove.

Most transformation systems provide control flow support. Some approaches allow non-deterministic rule selection. In general, this means that the result of the transformation can be different although the input model is the same. Informally, the global determinism property signifies that there is non-deterministic rule selection and the result of the transformation will be the same.

A pair of transformations, $G \Rightarrow^* H_1$ (a transformation sequence from G to H_1) and $G \Rightarrow^* H_2$, is confluent if there are transformation sequences $H_1 \Rightarrow^* X$ and $H_2 \Rightarrow^* X$. A graph transformation system is considered confluent if all pairs of transformations $G \Rightarrow^* H_1$ and $G \Rightarrow^* H_2$ in the graph transformation system are confluent. Often the weaker local confluence is stated, which requires that if there are two direct transformations $G \Rightarrow^{p1} H_1$ and $G \Rightarrow^{p2} H_2$, they can then be rejoined (meaning that there exist such transformation sequences $H_1 \Rightarrow^* X$ and $H_2 \Rightarrow^* X$) [Ehrig et al, 2006]. Local confluence is influential because of the role it plays in conjunction with termination. Based on Newman's lemma [Newman, 1942], it is proven that whenever a graph transformation system is locally confluent and terminating, it is also confluent.

Termination is generally, indefinite and must be established through the careful designation of rules, whereas local confluence can be shown for term rewriting and graph rewriting using the concept of critical pairs.

Critical pair analysis [Heckel et al, 2002] is used to check whether a rewriting system is confluent. Critical pair analysis has been generalized to graph transformations. This approach can be semi-automated as implemented in the AGG 2.0 tool [Runge et al, 2011]. The Critical Pair Lemma states that a term rewriting system is locally confluent if and only if all its critical pairs have common reducts [Huet, 1980]. For hyper-graph rewriting, the Critical Pair Lemma guarantees local confluence if each critical pair of a system has joining transformations that are compatible meaning they map certain nodes to the same nodes in the common reduct [Plump, 2005].

Confluence is an important aspect of graph transformation systems, because confluent (typed) graph transformation systems fulfill the requirements of global determinism: for each pair of terminating (typed) graph transformations $G \Rightarrow^* H_1$ and $G \Rightarrow^* H_2$ with the same source graph also the target graphs H_1 and H_2 are equal or isomorphic, where $G \Rightarrow^* H$ is called terminating if no (typed) graph transformation rule in the graph transformation system is applicable to H any longer [Ehrig et al, 2006].

Heckel et al. [Heckel et al, 2002] has shown how confluence can be ensured for typed attributed graph transformation systems. They have proved that an attributed graph transformation system is locally confluent if all its critical pairs are confluent.

Küster [Küster, 2006] has established a set of criteria to check for termination and confluence at design time through dynamic analysis of the transformation rules and the underlying metamodels.

3.3.3 Completeness and Mapping Property Class – PrC^{Comp}

The syntactic completeness property will soon completely cover the source language using transformation rules. Completeness assures that there exists a corresponding element in the target model for each element or structure in the source language.

Besides the mapping properties (bijection, injection, and surjection), there are classes of functions distinguished by the manner in which the source and target models are related/mapped to each other. In general, model transformation design tools do not include automatic methods to check these mapping properties on model transformations. Even if both the implementation of such algorithms and their execution require a reasonable amount of time and effort, in the case of complex transformations, these algorithms still cannot deduce the mapping properties. Thus, in most of the cases, the verification of these properties remains the responsibility of the transformation designer.

3.3.4 Semantic Correctness Property Class – PrC^{Sem}

It is worth mentioning that model transformations themselves can also be erroneous and therefore may hinder the quality of a model transformation-driven development and transformation-based verification and validation framework. Therefore, we have to deal with the model transformation itself in order to be free of semantic errors [Varró et al, 2003].

In theory, a straightforward correctness criterion would require to prove the correct mapping (e.g. semantic equivalence) of source and target models. However, as it was mentioned above, model transformations may define a projection from the source language to the target language (with deliberate loss of information) or transformation can add information independent from the source model into the target model. Therefore, semantic equivalence between models cannot always be proved. Instead, we define correctness properties (mapping) that are affirmative provided by the transformation. These correctness properties are transformation-specific, i.e. the mapping is domain-specific.

Most of the verification and validation approaches attempt to provide solutions for the preservation of the semantics, e.g. [Varró et al, 2003] [Giese et al, 2006] [de Lara and Taentzer, 2004] [Lengyel, 2006] [Barbosa et al, 2009] [Hulsbusch et al, 2010]. The semantic equivalence or semantics preservation of a model transformation ensures that the target model is semantically equivalent to the source model or that the important properties are preserved by the transformation.

The most well-known examples of behavior-preserving transformations are refactoring-like transformations. The term was introduced by Fowler [Fowler et al, 1999] to signify the restructuring of object-oriented programs, and since this innovation, several proposals for formalization and verification based on first-order logics and invariants have been created [Massoni et al, 2006].

Mens [Mens et al, 2002] presented the first formal approach to refactoring based on graph transformations, where the focus relied on the analysis of conflicts and dependencies among rules. A survey regarding software refactoring is elaborated in [Mens and Tourwe, 2004]. Based on our classification framework, this mapping holds special significance: each refactoring-like transformation has a domain-specific mapping that defines the properties, which are preserved by the transformation.

Although there are many different types of transformations that are useful in model-driven development, each transformation preserves certain aspects of the source model in the transformed target model. The properties that are preserved can differ significantly depending upon the transformation type. In regards to refactoring or restructuring, the (external) behavior must be preserved, while the structure is modified and with refinements, the preservation of the program correctness is paramount. The technical context also heavily influences that which requires preservation. For example, in the case of a database transformation, it is necessary to preserve the integrity of the database; while in the case of a program

transformation; syntactic well-formedness and the type correctness of the program are of the utmost importance. These are all examples of domain-specific mapping. According to our classification framework, a relevant part of these transformations (both domain and transformation properties) can be verified in the semantic domain, either on the transformation image or based on the correspondence between the source and target models.

The work related to semantic correctness criteria of model transformations is very limited. The preservation of certain properties by the transformation is discussed by Varró et al. [Varró et al, 2003]. However, only a few approaches exist (e.g. [Assmann, 2000]) to analyze the semantic correctness of arbitrary model transformations, when transformation-specific properties are targeted for verification.

Varró et al. [Varró et al, 2003] presented a partly-automated, language-independent, modeling framework to provide formal verification through model checking. This framework examined model transformations from an arbitrary, well-formed model instance, of the source modeling language, into its corresponding target preserves language-specific properties.

The graph rewriting-based model transformations cover a wide-range of model processing. These transformations implement a variety of semantic mappings. Verification and validation approaches currently provide some specific mappings (e.g., equivalence preserving or bi-similarity in specific domains). These mappings are developed individually, in order to extend them or to create new ones requires a deep understanding of the verification/validation approach. Therefore, this field requires further research and development to bring this approach to a wider audience (model transformation designers) in order to further development in configuring the verification/validation related issues of model transformations.

3.3.5 Attribute Range Property Class – PrC^{Attr}

The attribute value property class is comprised of the attribute value and interval validation. These properties can be expressed in first-order logic, extended with traversing capabilities. Example languages for defining attribute value and interval conditions are OCL, C, Java, and Python.

Aside from basic structural conditions (PAC and NAC [Ehrig et al, 2010]), several approaches apply constraint languages to define the execution conditions and requirements related to the results. These conditions and requirements are pre- and post-conditions of transformation rules.

A precondition assigned to a transformation rule is a Boolean expression that must hold true at the moment the transformation rule is fired. A post-condition assigned to a transformation rule is a Boolean expression that must hold true following the completion of a transformation rule. If a precondition of a transformation rule is not true, then the transformation rule fails without being fired. If a post-condition of a transformation rule is not true following the execution of the transformation rule, the transformation rule fails.

As was stated earlier, the execution of a transformation rule is as follows: (i) Finding the match according to the structure of the LHS. (ii) Validating the constraints defined in LHS on the matched parts of the input model. (iii) If a match satisfies the constraints (preconditions), then fires the transformation rule, otherwise the rule fails. (iv) Validating the constraints, defined in RHS, on the modified/generated model, i.e., if the result of the transformation satisfies the post-conditions, then the rule was successful, otherwise the rule fails.

A direct corollary is an expression in LHS represented as a precondition to the transformation rule, and an expression in RHS is a post-condition to the transformation rule. A transformation rule can be fired if and only if all conditions established in LHS are true. Moreover, if a transformation rule completes successfully, then all conditions enlisted in RHS must be true [Lengyel, 2006].

The definition of an attribute value and interval condition are supported e.g., in AGG (Java), AToM^3 (Python), GReAT (OCL), Fujaba (Java), VIATRA2 (Java) and VMTS (OCL) tools. Application conditions which are checked after the execution of a rule are supported by the approaches similarly to preconditions. However, the semantics can differ: while in AToM^3 the condition is checked only, AGG, VIATRA2 and VMTS provide a rollback mechanism if the condition is not met.

3.3.6 Architectural Property Class – *PrC^{Arch}*

Architectural properties are invariants, typically, reachability and topological properties of the model. These are output, model-related properties. An invariant property should always hold on the model elements. Every execution path, and therefore every rule of the model transformation, must guarantee that invariant properties are always satisfied. An example invariant property is the following: Assuming a water tank simulation system - we require that the temperature of the water in a specific water tank always be between 45 °C and 90 °C. In order for the chemical process to be complete, the water must be at least 45 °C. However, if the water temperature reaches 90 °C or above, the water tank will be damaged. Therefore, each of the transformation rules must ensure the water temperature remains within this range.

A reachability property describes a desired scenario, which can be reached through at least one execution path. This means that there exists a sequence of rule executions that transport the input model into the desired state. Examples of this are Statechart and Petri net processing transformations. A reachability analysis can check whether a desired Statechart or Petri net configuration could be achieved using a model transformation. Alongside this, the verification should account for the input model, or the class of input models, for which the transformation provides the reachability properties.

The reachability of configurations allows us to check whether a given configuration can be reached through a finite set of transformation rules. Many verification problems can be formulated as the reachability (or non-reachability) of any given configuration of the system. Built around the technique of graph parsing, one can decide whether the target configuration can be generated by the graph transformation system, if beginning from a specific, initial model, thus providing the means for backtracking reachability analysis [Baresi et al, 2003].

Both GROOVE [GROOVE] and CheckVML [Rensink et al, 2004] model-checker tools facilitate in verifying invariant and reachability properties.

3.3.7 Summary

This section has introduced and discussed the graph rewriting-based model transformations related property classes. These property classes have been motivated by certain verification and validation related questions. The property classes provide the answer to the question: what can be and what needs to be verified/validated by model transformations. The property classes also make possible to classify the existing approaches according to these verification and validation questions, i.e., (i) to define a framework for comparing individual verification and validation approaches and tools, and (ii) to identify and evaluate approaches and tools for a specific model transformation activity that requires verification and validation capabilities.

We believe that our property classes contribute to efficient methods that provide comfortable ways of verification/validation of model processors.

Figure 3-6 depicts a summary of the classification. This figure is an overview that can be used as a quick guide related to the three views of the classification framework. The figure is assembled based on the analysis and evaluation of more than 30 approaches and tools. Following a single column, we can

identify the functionalities of an individual tool or approach. Column groups (*Graph Transformation Approaches, Graph Transformation Tools,* and *Model Checker Tools*) facilitate to overview the key areas of a certain group. E.g., regarding to In Figure 3-1, the group *Model Checker Tools* belongs to the $Path^{TransTS}$ category of the *Path View* and the *Static* category of the *Computational View*. Following a single line, we can collect the tools and approaches supporting a specific attribute regarding a certain view. For example, following the *Dynamic* category of the *Computational View* we can identify the tools and approaches that provide dynamic type verification and validation. Furthermore, row groups (*Path View, Computational View,* and *Property View*) make it possible to identify both the well supported and the missing features. E.g., the PrC^{Synt} and PrC^{Comp} properties in the *Property View* are well covered by the tools and approaches, while only some of the examined tools and approaches belong to the $Path^{Hybrid}$ category of the *Path View*.

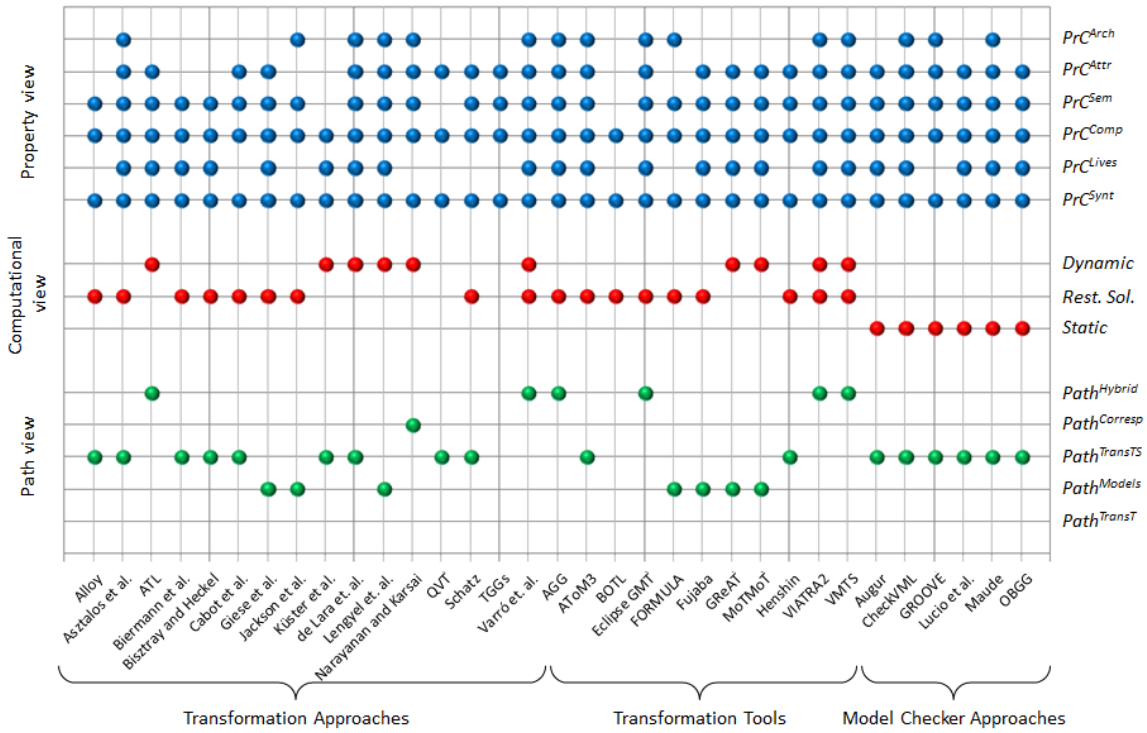


Figure 3-6 Classifying Model Transformation Approaches by Model Processing Properties – Summary of the *Property View*, *Computational View* and *Path View*

3.4 A Method for Taming the Complexity of Model Transformation Verification/Validation Processes

This section introduces the concept of taming the complexity of the verification/validation solutions by starting with the most general case and moving towards more specific solutions [36].

Several static approaches provide formalism and verify that the semantics are preserved or guaranteed during the transformation of a model, e.g. approaches provided by Asztalos et al. [5], Biermann et al. [Biermann et al, 2011], Bisztray et al. [Bisztray et al, 2008a], Cabot et al. [Cabot et al, 2010], or Schatz [Schatz, 2008].

The approach of Asztalos et al. focuses on the static analysis of special model processing programs. This approach provides the theoretical basis for a possible verification framework. It applies a final formula that describes the properties that remain true at the end of the transformation. It is possible to derive either proof or refutation of a verifiable property from this final formula. The approach provides predefined components to deduct the desired properties.

In the approach presented by Bisztray et al., Communicating Sequential Processes (CSP) are applied to capture the behavior of processes both before and after the transformation in order to understand and control the semantic consequences. The approach verifies semantic properties of the transformations at the level of rules, such that every application of a rule has a known semantic effect.

In the approach of Bierman et al., model transformations are defined as a special kind of typed graph transformations. The solution implements a formal approach to validate various functional behaviors and consistencies of model transformations.

There exist noticeable differences between the complexity of static and dynamic verification and validation approaches. The static technique is more general, because its responsibility is to determine if the rule-based system itself meets certain requirements. Contrarily, in the case of the dynamic approach, the transformation is analyzed based on a single specific input.

This is a common knowledge that the algorithmic complexity of verification and validation can be quite challenging, if not altogether hopeless in a general sense. However, practical cases do not require generality. The complexity-related questions are as follows: Does the problem contain specific subclasses that are solvable, yet practically relevant? Is it necessary to analyze the algorithm of the transformation system? Will it be sufficient to verify the system for a certain class of possible input models?

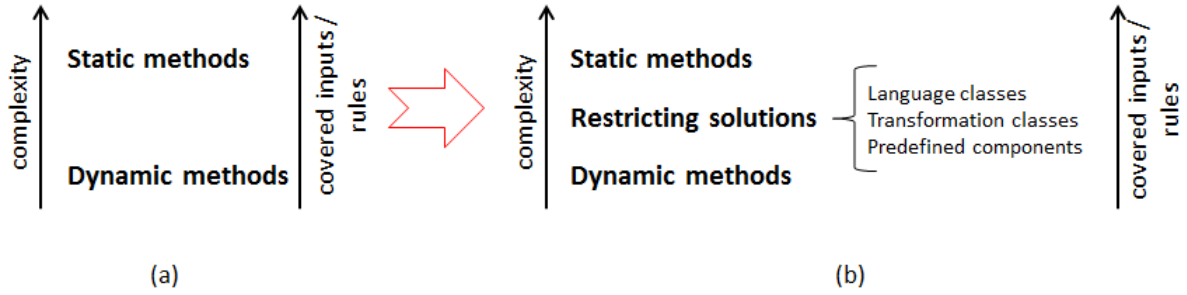


Figure 3-7 Taming the complexity of model transformation verification/validation processes

Our classification says that the static approach is the most general and the dynamic is the most specific of verification and validation methods (Figure 3-7a). In order to reduce verification and validation complexity, we classify the verification and validation approaches from the complexity point of view. In order to accomplish this, we begin with the general case (static verification) and create more specific cases (dynamic verification). Between these two extreme approaches, we identify several complexity-related restricting solutions (Figure 3-7b). These methods do not attempt to prove the semantic correctness for one or all possible inputs (prove the properties of a transformation system), but instead take a class of input types into consideration. We have identified the following complexity categories:

A. Static methods
B. Restricting solutions:
    1. Language classes
    2. Model transformation classes
    3. Predefined components
C. Dynamic methods

A. Static methods. Model checker tools (e.g. Augur [König and Kozioura, 2008], CheckVML [Rensink et al, 2004], or GROOVE [GROOVE]) apply static methods during the verification.

B1. Restricting solutions – Language classes. This approach defines a class of input models. Based on a metamodel, a language class is defined by additional metamodel constraints or the simplification of the metamodel, i.e. through the elimination of some domain concepts. As a result, the modified language contains only a restricted class of original models, therefore, the complexity of the processing transformation decreases along with the complexity of the transformation verification. Examples of language classes are provided in OMG Query/View/Transformation Specification [OMG QVT, 2016]. The Annex A of the QVT specification introduces two language classes: *Simple UML Metamodel and Simple RDBMS Metamodel*. These domains provide limited language elements and attributes that are suitable for defining the required models, but do not provide additional, unnecessary language constructions. For instance, a *Table* containing *Keys* can be modeled, but the *Key* type does not provide attributes to specify further details. Another example is the limitation of the multiplicity to *1* or *0..1*. A third example presents itself when only finite input models are permitted. A sample language class was introduced earlier in this chapter: *DomainServers* (Figure 3-2).

B2. Restricting solutions – Model transformation classes. This approach restricts the rule specification language itself. We modify the metamodel of the rule specification language in order to allow for transformation definitions with specific properties. An example of a transformation class is one in which rule chains are allowed (successively applying several rules in a predefined order) but loops are forbidden. Proving the termination of such transformation requires reasonably less complexity than in the general case, when loops are permitted. For example, in the field of layered grammars [Ehrig et al, 2005], Bottoni et al. [Bottoni et al, 2000] developed a termination criterion which ensures that the creation of all objects of a certain type should precede the deletion of an object of the same type. Therefore, the layer deleting an object of a given type should not create such an object, nor should the subsequent rules. This means the productions in a deletion layer will terminate. Therefore, the termination analysis of transformations satisfying this criterion requires less complexity than the general case.

B3. Restricting solutions – Predefined components. In this case, the verification procedure is constructed from predefined components. We can state facts about the components, which the verification process treats as axioms, therefore, the results of other tools or human analysis can be also utilized. Applying these predefined components, we can deduce what output model properties are provided by the given transformation for the provided input domain. For example, the formal language, developed by Asztalos et al. [5], is able to express a set of model transformation properties. The language is appropriate to specify both the properties of the output models and the properties of the relations between the input and output model pairs. In most cases, the proofs within the class of predefined components are conducted by dedicated checker tools (e.g. GROOVE [GROOVE] or CheckVML [Rensink et al, 2004]) or through human analysis.

C. Dynamic methods. Examples of dynamic methods are provided by Lengyel [Lengyel, 2006]. In their approach, the validation of a transformation system is achieved with constraints assigned to the rules as pre- and postconditions. A similar approach was developed by Narayanan and Karsai [Narayanan and Karsai, 2008], in which the semantic equivalence between inputs was guaranteed via bi-simulation checks on the execution log of the transformation.

As a conclusion for the topic *taming the complexity of model transformation verification/validation processes*, we can say that applying these restricting solutions, i.e. working with language classes, transformation classes, or predefined components, we ensure that (i) the verification or validation of

model transformations, including transformation systems, requires less complexity than the classical static verification and (ii) the verification or validation results are valid not only for a specific input, but for a class of input models or transformation classes.

3.5 Test-Driven Verification/Validation of Model Transformations

This section discusses a method and algorithms for test-driven verification/validation [7].

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to the process of executing a program or application with the intent of finding software errors or other limitations. Software testing can be stated as the process of validating and verifying that an artifact (i) meets the requirements that guided its design and development, (ii) works as expected, (iii) can be implemented with the same characteristics, and (iv) satisfies the needs of stakeholders. Model transformations are also software artifacts; therefore, model transformation testing methods are also based on the widely used general testing principles.

Testing can never completely identify all the defects within software [Pan, 1999]. Instead, it compares the state and behavior of the artifact by which someone (the software engineer or the domain specialist) might recognize a problem [Leitner et al, 2007].

Testing model transformations is any activity aimed at evaluating an attribute or behavior of a model processor and determining that it meets its required results. The difficulty in testing of model transformations stems from the complexity. Testing is more than just debugging the execution of the transformation. The purpose of testing are quality assurance and verification/validation [Hetzel, 1988].

A reasonable part of the defects in transformations are design errors. Bugs on software artifacts, including model transformations, will almost always exist in any software module with moderate size. This is not because architects, engineers and programmers are careless or irresponsible, but because the complexity of software artifacts is generally hard to manage. Humans have only limited ability to handle it. It is also true that for any complex system, design defects can never be completely eliminated [Kaner, 2006].

Because of the complexity, discovering the design defects in model transformations is difficult. All the required properties need to be tested and verified, but complete testing is infeasible. A further complication has to take into account that is the dynamic nature of software artifacts. If a failure occurs during preliminary testing and the design is changed, the transformation may now work for a scenario that it did not work for previously. However, its behavior on pre-error scenarios that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often too high.

Regardless of the limitations, testing is an integral part in model transformation development. In our context, testing is usually performed to improve the quality and to verify/validate transformations. Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances.

Testing is heavily used as a tool in the verification and validation process of software artifacts. We cannot test quality directly, but we can test related factors to make quality visible.

Tests with the purpose of validating the model transformation works are named clean tests, or positive tests. The drawbacks are that it can only validate that the transformation works for the specified test cases. A finite number of tests cannot validate that the transformation works for all situations. On the

contrary, only one failed test is sufficient enough to show that the transformation does not work. Testing quality of software artifacts can be costly, but not testing software artifacts is even more expensive.

In summary, the primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product works properly under all conditions but can only establish that it does not function properly under specific conditions [Kaner et al, 1990]. The scope of model transformation testing often includes examination of the transformation definition, execution of that transformation in various conditions as well as examining the aspects of the transformation: does it do what it is supposed to do and do what it needs to do. Information derived from testing may be also used to correct the process by which the transformations are developed [Kolawa and Huizinga, 2007].

The goal of the test-driven validation approach is to test graph rewriting-based model transformations by automatically generating appropriate input models, executing the transformations and involving domain specialists to verify the output models based on the input models. It is important that the semantic correctness of the output models cannot be automatically verified, i.e. we need the domain specialists during both the transformation design and testing.

The test-driven validation method needs a model transformation definition and the metamodels of both the input and output domains. The method is about to automatically generate input models that cover all execution paths of the transformation. Covering the whole transformation means that each of the rules in the transformation will be executed at least ones. Furthermore, each of the decision points (branching points, forks) are evaluated for both the *true* and the *false* branches, i.e. all of the paths in the control flow model are traversed. The generated input models represent a set of input models. We use the expression *set* for a bunch of input models that cover the whole transformation. The number of models in the sets can vary based on the actual domain and on the actual transformation definition. An objective of the solution is to make these model sets minimal, which means to minimize the number and the size of these models.

The method should generate such typical models, which effectively cover the whole transformation. We execute the transformation for these input models, then we involve domain specialists. We provide the input model and output model pairs to the domain specialists. Then, based on the input and outputs, and not taking into account the transformation definition, they can decide whether the transformation does the right processing. Without the domain specialists, we cannot verify that the output model is really correct, i.e. which is the appropriate output for a given input model.

As we have already mentioned, the input model sets should cover the whole transformation, therefore, the main goal of the approach is to minimize the possibility that the transformation works perfectly for N input models, but fails for the $N+1^{th}$ input model. What is even worse: generates the output for the $N+1^{th}$ input model, but the output is not the expected one, i.e. there is a conceptual error within the transformation definition.

Figure 3-8 introduces the architecture of the approach. Input model sets are automatically generated based on the input metamodel (*Metamodel A*) and the transformation processes (*transform*) including the transformation rules and the control flow model. The output models should instantiate the output metamodel (*Metamodel B*). Finally, domain specialists verify the correspondence between the input and output models (*corresponds*).

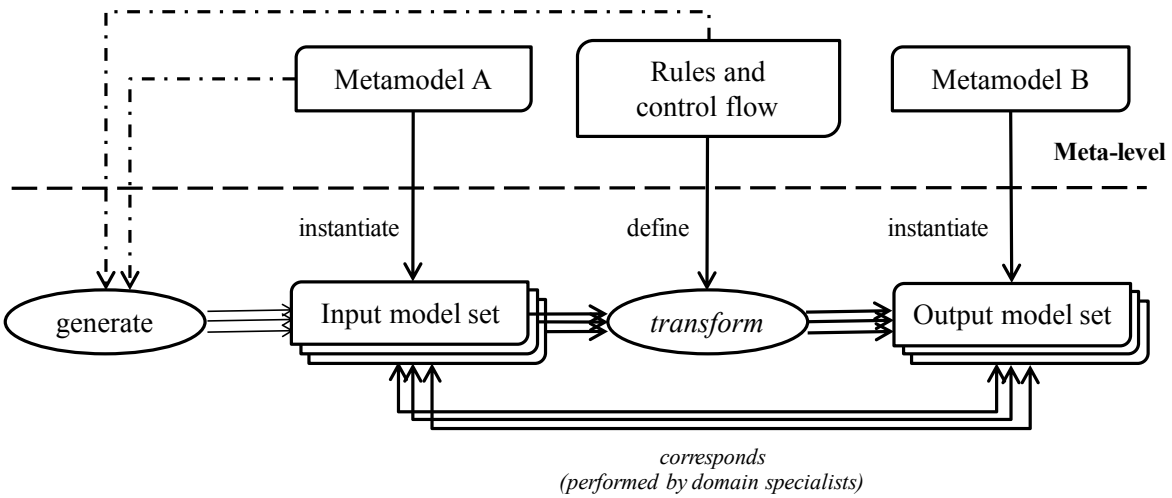


Figure 3-8 A test-driven method for validating model transformations

Scenarios that are targeted to be supported by the test-driven validation approach are as follows:

- Automatic generation of valid input models that support the testing of the model transformation. The generation is based on the metamodel of the input domain and the transformation definition (control flow model and the transformation rules).
- Automatic generation of valid input model sets that cover the whole model transformation, i.e. executing the transformation with an input model set means that all of the transformation rules will be executed, and all of the paths in the control flow model are traversed.
- Automatic generation of a valid and minimal input model set that covers the whole model transformation.
- Automatic generation of valid input models that support the testing of one or more selected transformation rules, i.e. executing the transformation with these input models means that the transformation rules to be tested will be executed. However, other transformation rules of the control flow model can be skipped in this scenario, e.g. certain branches or loops of the whole transformation can be omitted. The main goal of this scenario is the debugging of the selected transformation rules.
- Automatic generation of a valid and minimal input models that support the testing of one or more selected transformation rules (e.g. a selected sequence of transformation rules within the whole transformation definition).

Addressing the above scenarios, the test-driven validation approach can support the verification/validation of graph rewriting-based model transformations.

During the analysis and implementation of the above scenarios we have to take into account the following elements and aspects of model transformations and transformation rules:

- In order to cover all of the transformation rules, all LHS patterns should be either present in the generated input model or should be established during the transformation execution before reaching the rule requiring the pattern.
- The method should take into account the modifications performed by the rules. Rules can also delete or break LHS patterns prepared for other rules. In addition, rules can prepare LHS

patterns for other rules performed later. Therefore, deletion and creation of nodes and edges, furthermore, the attribute value modification should also be considered.

- The control flow model of the model transformation has an effect on the processing. Not only rule sequences, but also the effects of the conditional branches and the loops should be considered.
- Different treatment is required by the in-place transformations and the transformations generating a separate output model. We should take into account whether the transformation modifies the input model.
- The generated input model is ideally connected, but it is not a strict requirement. This depends on the actual domain and on the metamodel of the domain.

We have worked out two versions of the test-driven validation approach: the *Basic* and the *Advanced* versions.

The *Basic* algorithm takes into account the following aspects of model transformation definitions:

- The transformation rules that should be covered by the generated input model.
- The LHS structure of the concerned rules.

The *Advanced* solution extends it with the following considerations:

- Collects the RHS patterns of the processed rules in a global store, and takes into account both the actually generated input model and the RHS patterns of the already processed rules when decides whether the LHS pattern of the next rule could be found in the processed model at a certain point of the model processing.
- Takes into account rule sequences and their operations (node and edge deletion, creation and attribute modifications).
    - The solution applies rule concatenations to calculate the resulting RHS patterns at a certain point of the transformation. Rule concatenation means to contract two rules in order to derive one transformation rule which behavior functionally replaces the application of the two original rules. The concatenation results a new rule with a new LHS and RHS pattern. The calculated RHS pattern is also taken into account when the method searches the LHS of the next patterns.
    - Includes the conditional branches, therefore, considers the possible execution paths of the transformation. This is also supported by the rule concatenation and can result different rule execution sequences.
    - Takes into account the loops of the control flow definition.

The GENERATETESTINPUT-BASIC algorithm gets the transformation definition, the collection of the concerned rules and the input metamodel as parameters. Initializes a model based on the input metamodel. This model is built by the following part of the algorithm. The core part of the algorithm is a loop that takes the next transformation rule based on the control flow model of the transformation and the collection of the rules that should be covered by the generated input model. Next, the algorithm checks if the LHS of the actual rule is already present in the generated model. If not, then clones it and attaches the copy of the LHS to the input model under generation. This method, attaching the LHS of the actual rule, can happen in different ways. In the case of the basic algorithm, we search a common node based on the metatype of the node, and we attach the new pattern utilizing this common point.

Necessarily, this step considers the definition of the input metamodel in order to the generated model be a valid instance of the metamodel. This step means that generating valid instances of the input metamodel requires that the LHS structures of the transformation rules, that are attached to the generated model, be valid partial instances (Section 5.2) of the input metamodel. This allows that the attached model part, in case it is necessary, could be extended to a valid instance model by the further steps.

Algorithm. Pseudo code of the GENERATETESTINPUT-BASIC algorithm

```
00: GENERATETESTINPUT-BASIC(Transformation T, Collection RuleCollection, Model InputMetamodel): Model
01: Model InputModel = INITIALIZEMODEL(InputMetamodel)
02: while (Rule rule = T.GetNextRule(RuleCollection)) do
03:   if not OutputModel.ContainsPattern(rule.LHS) then
04:     Model temporaryPattern = CLONEMODEL(rule.LHS)
05:     InputModel.ContainsStructure(temporaryPattern)
06:   end if
07: end while
08: return InputModel
```

The GENERATETESTINPUT-ADVANCED algorithm extends the basic algorithm with the following steps:

- Stores the RHS patterns of the processed transformation rules in the *RHS-Store*. Furthermore, the LHS of the actual rule is searched not only in the actual version of the generated model, but also in the *RHS-Store*.
- The CALCULATERHSPATTERNVARIATIONS method applies rule concatenation technique and calculates the different RHS pattern variations. The method gets the transformation, the actual rule and the RHS patterns from the *RHS-Store* to utilize them during the calculation of the pattern variations.
- The CALCULATERHSPATTERNVARIATIONS method also takes into account both the conditional branches and the loops of the transformation definition.

These techniques of the GENERATETESTINPUT-ADVANCED algorithm make possible to generate minimal model sets that support the testing of the whole transformation. This means that the techniques help to minimize the number and the size of the generated models.

Algorithm. Pseudo code of the GENERATETESTINPUT-ADVANCED algorithm

```
00: GENERATETESTINPUT-ADVANCED(Transformation T, Collection RuleCollection, Model InputMetamodel): Model
01: Model InputModel = INITIALIZEMODEL(InputMetamodel)
03: PatternStore RHS-Store = INITIALIZEPATTERNSTORE()
04: while (Rule rule = T.GetNextRule(RuleCollection)) do
05:   if not InputModel.ContainsPattern(rule.LHS) && not RHS-Store.ContainsPattern(rule.LHS) then
06:     Model temporaryPattern = CLONEMODEL(rule.LHS)
07:     InputModel.ContainsStructure(temporaryPattern)
08:     RHS-Store.AddPattern(rule.RHS)
09:     Pattern[] RHS-PatternVariations = CALCULATERHSPATTERNVARIATIONS(T, rule, RHS-Store)
10:     RHS-Store.AddPatterns(RHS-PatternVariations)
11:   end if
12: end while
13: return InputModel
```

The presented algorithms address the above requirements, i.e. applying these algorithms we can generate valid input models that support the testing of one or more selected transformation rules. Utilizing these algorithms with different input parameters, we can also generate valid and minimal input model sets that

cover whole model transformations. The details of certain parts of the algorithms, e.g. the get next rule of the transformation (taking into account the branches and the loops), the pattern search in the generated model and in the *RHS-Store*, and the CALCULATERHSPATTERNVARIATIONS method, can be implemented in different ways. This also means that further optimization can be introduces, e.g. with the application of various heuristics.

3.6 Conclusions

Important semantic information can easily be lost or misinterpreted in a complex transformation due to errors in the transformation rules or in the processing of the transformation. Methods are required to verify that the semantics used during the analysis are indeed preserved across the transformation. Automation certainly increases the quality of model transformations as errors manually implanted into transformation programs during implementation are eliminated. Verification and validation of model transformations is required, which assures that conceptual flaws in transformation design do not remain undetected.

This chapter has emphasized the necessity of verification/validation methods that increase the quality of model transformations and help to ensure that model transformations perform what they are intended to do. Focusing on the graph rewriting-based model transformations, we have discussed the different scenarios of model transformation verification and validation.

We have reviewed that rule-based systems can effectively automate problem solving standards. Such systems provide a method for capturing and refining human expertise and affirm their relevance to the industry. Instead of representing knowledge in a relatively declarative way, i.e., numerous things that are known to be true, rule-based systems represent knowledge in terms of a collection of rules that tell what should be done, i.e., what can be concluded from different situations? The motivation of this area was to support the verification/validation of rule-based systems.

We have provided a method, which facilitates to apply the graph rewriting-based dynamic validation results in the field of rule-based systems. The solution facilitates to validate single rules, rule chains, and whole transformations as well. The validation is driven by the pre- and postconditions assigned to the rules.

We have introduced and discussed the graph rewriting-based model transformations related property classes. These property classes have been motivated by certain verification and validation related questions. The property classes provide the answer for what can and what needed to be verified/validated by model transformations. The property classes also make possible to classify the existing approaches according to several verification and validation questions.

Next, we have introduced the concept of taming verification complexity. We have seen that the static validation method is more general and raises challenges that are more complex. We have discussed the possibilities of reducing the complexity of verification/validation and have introduced different restricting solutions.

Then, we have provided our dynamic validation method, and we have introduced the key motivation and challenging points of the test-driven validation approach. In addition, we have provided both the *Basic* and the *Advanced* version of our solution that makes possible to generate the test input models for model transformations.

The discussed area related selected scientific results from my research activities are summarized in Section 7.1 (Thesis I: Methods for Verifying and Validating Graph Rewriting-Based Model

Transformations). I believe that my results and novel solutions contribute to the way to reach methods that provide efficient solutions for verification/validation of model processors.

4 Model-Driven Methods Based on Domain-Specific Languages and Model Processors

4.1 Introduction

The utilization of model-driven methods and solutions supports the clarity and efficiency of requirements analysis, contributes to the clear project scope definition, and ensures the quality improvement of software products. The ultimate goal is to reach customer satisfaction and increase the success rate of software projects [Davies, 2011] [Norbisrath, 2013].

Software development requires adequate methods for requirements engineering, design, development, testing and maintenance. The more complex the system is, the more sophisticated methods should be applied. A significant part of software projects is short on appropriate requirements engineering, communication, development and testing method, furthermore, verification and validation processes. In summary, not the right method is applied, and the project turns into ad-hoc design and development decisions. This chapter introduces model-driven methods that assure model-driven requirements engineering, support the creation and management of domain-specific languages, helps in model processing and model transformation. The method is based on the modeling of the software requirements in a way that these models can be used to automatically generate several artifacts during the engineering process. [8]

As a further contribution to the above goals, we discuss a method that helps designing and managing domain-specific languages and models; we introduce a possible way to transparently switch between the textual and visual views of semantic models, furthermore, techniques are also provided how model-driven methods are applied in software development processes.

4.2 Quality Assured Model-Driven Requirements Engineering and Software Development

Developing software artifacts in enterprise environments should be based on mature methods. Software products drive almost all parts of our life. However, software development methods and the methods supporting the whole process still can be made more adequate. Appropriate approaches can efficiently support unambiguous requirement definition, capture and define user processes, allow effective development that results in quality products, and verified/validated requirements. Adequate requirement engineering and analysis [Sommerville and Kotonya, 1998] [Pohl, 2010] can define the obvious project scope and result in the agreement between the customer and the developer team during the project and also in the end when deliverables should be adjudged and accepted. The unambiguous requirement specification, being formal enough, can drive the whole software project, including the development, testing, documentation generation, and maintenance as well. Furthermore, it is still important to be able to accept the continuous changes and run an iterative agile process cost effectively [Agile Manifesto, 2001].

There are several opportunities in the system development that motivates software architects and developers to work out and use different methods. Notable motivating issues are the problem of informal requirements specifications and the consequences of the ad-hoc project scope. These issues can result in misunderstandings and losing the control over the project. We have assembled a few points that further drive our activities and motivate us to develop more advanced and adequate techniques in the development process:

- *Inadequate requirements analysis method*: The relevant customer processes are not or under analyzed. Consequently, they are not addressed properly by the provided solutions.

- *Ambiguous scope definition*: A textual description of the requirements can easily mean different things for the customer and the solution providers. Without languages that are formal enough and understandable both for the customer, i.e. domain experts, and the software experts, the scope of the project and the real requirements can become unclear.
- *Only software functions are defined but domain processes are not analyzed*: This issue assumes that the software drives the business and not the business requirements define the software functions.

From the beginning, which goes back to the previous decade, our vision was to develop software systems with a methodology that targets and supports the followings:

- Techniques supporting effective requirements analysis that ends with formal user stories, which can be validated by the customer. This means that the customer understands the specification, because it uses the domain concepts and defines their business processes.
- Domain-specific languages that are applied to define use cases, user stories (the business processes that are required to be supported), domain dictionary, and the high-level requirements. The languages facilitate cross-references between the model elements, i.e. referring domain concepts and requirements from use case models and activity models (user stories).
- A method to build the semantic model based on the business processes defined by the user stories.
- A technique to generate as much of the source code as possible from the requirements specification. The generated code is based on supporting libraries and a framework that are continuously developed by senior developers. The generated code covers those parts of the implementation that are very similar in each project, e.g. data management (database-related functionalities, concurrency management), communication between the client and the server. Of course, there can be custom logic, which is hard and therefore is not worth modeling, consequently, the related code is not generated.
- Each requirement is connected to the model elements that define the software development activity even if it is generated or implemented manually. This makes possible to follow the modifications and apply the required changes during the whole lifecycle of the system.
- A technique to automatically generate relevant test scenarios for the manually written modules based on the formal user stories.
- A technique to automatically generate formal but user readable documentation, i.e., the documentation is also driven by the identified user stories, automatically generated and validated by the customer. The documentation includes the detailed specification that is easy to read and validate by the end-users. Based on the documentation, the representatives of the customer can decide if the specification meets their business requirements.
- A method to ensure the continuously up-to-date state of the semantic model (requirements specification) all along the lifecycle of the given solution.

We have worked out a domain-specific modeling and model processing-based method for supporting effective software development. The method covers the whole development process, including requirements engineering, development, testing and documentation as well. Figure 4-1 introduces the main concept of the suggested requirements engineering and development method. The requirements engineering and analysis result in the *Requirement specification*, which is a collection of both human

readable and formal models. This means that these models can be easily understood by domain specialists (customers) and also formal enough to serve the software artifact generation. The specification includes use cases, activity (process definition) and further models collecting the user requirements. These models define the high-level requirements, the domain vocabulary, the use cases and the detailed user stories. The process definition models are step-by-step scenarios defining the exact algorithm (workflow) of the business processes that should be covered with the solution. Based on the requirements specification we generate the software artifacts, the relevant test scenarios and documentation as well. The testing performed on the executables is based on the automatically generated test scenarios. The verification and validation, performed on the generated and manually extended software products, is also based on the requirements specification.

These all mean that the well-defined requirements specification drives the whole process. This is one of the most important points of the entire method, i.e. the requirements specification is in the focus: the customer and the developer team agree on the scope of the project with the help of the requirements specification models. Next, this specification drives the whole process: we always go back to the specification and derive each of the required artifacts based on it. The method provides an iterative process: the continuously gained experience and the feedback improve the requirements specification and therefore influence the next iteration as well: newly generated and altered software artifacts, test scenarios, and the refreshed documentation.

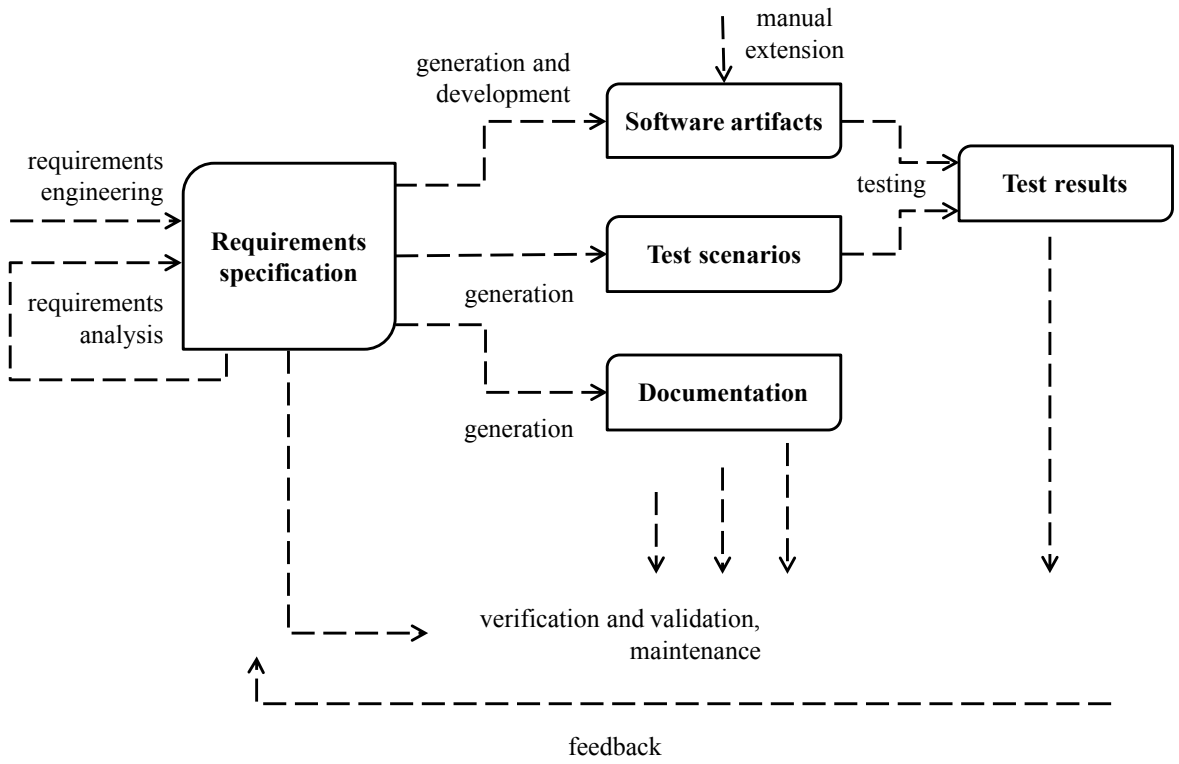


Figure 4-1 Assuring the quality of software development projects with model-driven techniques

Now let us discuss how the method improves certain aspects of the software engineering process. Our overall goal with the method is to *increase the quality of the produced artifacts*, *the effectiveness of the requirements analysis*, *the unambiguity of the project scope and development*, *and the customer satisfaction*. The scope of the method is as follows:

- To identify, which characteristics of the system to be developed, determine if the suggested method could be applied for it. By system characteristics we mean the size of the system under

development, the architecture (e.g. web client, server, database layer), the technology (e.g. Java), and the development method (e.g. waterfall, agile, other). We examined numerous software development technologies and methods to be able to provide as generic method as it is effectively necessary.

- To identify the target audience. The key is in the length of the lifecycle. The longer the software system is maintained, the more changes must be applied during the lifecycle. By having a well-understandable software model, the changes can be introduced at a significantly lower cost. We believe that this affect can produce shorter ROI (Return of Investment) period if the method and the tooling are harmonized. Our experiences show that our methodology aims bigger systems, which require careful modeling, exact processes and workflows, furthermore detailed documentation and deep testing. Such target audience can be banking applications, civil service, public administration, insurance companies, medicine factories, etc.
- A method supporting the definition of software specifications on the level of abstraction high enough for the domain experts and the further engineering process. The method and the tools should produce the documentation and the basement of the following development activities. Therefore, it provides a commonly used communication platform between the participants of a software development project.
- The method and the provided supporting tools should be effective enough to make it easier to follow the process than to sabotage it. In general, the overall goals of software projects are well known, and every software expert understands their needs and agrees with them. However, they still follow the "ad-hoc" approach, usually saying that we do not have enough time to be systematic. We believe that an effective method and the appropriate tool support can break this habit.

4.2.1 Domain-Specific Languages for Requirements Engineering

Requirements engineering [Sommerville and Kotonya, 1998] is the process of formulating, documenting and maintaining software requirements. Requirements engineering has a significant role in successful software engineering processes. Requirements analysis in software engineering encompasses those tasks that go into determining the needs or conditions to meet for a new or altered software product or artifact, taking into account the possibly conflicting requirements of the various stakeholders, analyzing, documenting, validating and managing software and system requirements.

The IEEE Standard Glossary of Software Engineering Technology [IEEE Standard Glossary, 1990] defines a *software requirement* in the following way:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

We have developed four DSLs and integrated their usage, which provide an extensive toolset to facilitate the description and collection of the requirements in an efficient way:

- *Use Case DSL*: to describe the actors and the use cases of software systems,
- *Activity DSL*: to describe the workflows (user stories) related to specific user and test case scenarios,

- *Requirements DSL*: a dictionary of the requirements to make it possible to reference them from other models,
- *Concept DSL*: a glossary of domain related concepts that can be referenced from other model elements.

The tool support of the method is implemented in the Eclipse Framework [Eclipse] as an Eclipse plugin and further model processing components. We discuss the metamodels of the domain-specific languages, introduce the plugin, which is based on the Eclipse Graphical Modeling Framework (GMF) within the frame of Eclipse Graphical Modeling Project (GMP) [Eclipse GMP]. Furthermore, we share some details about the realized editor using the Eclipse Extended Editing Framework [Eclipse EEF].

The domain-specific languages not only support the quality assured model-driven requirements engineering method, but also serve as a basis for generating software artifacts. First, we introduce the common elements of the four modeling languages, and then present the concept of each language separately. Next, the tool support for the overall method is detailed. Finally, the processing of the models, i.e. the methods that generate code, documentation and test scenarios are addressed.

4.2.1.1 Common Elements of the DSLs

The four DSLs have a common tree view editor interface, a table view and a more intuitive graphical view for visual editing. These DSLs are an improved and expanded version of the ubiquitous UML use case and activity models [OMG UML, 2015]. We have modified the original UML languages, because we wanted them to fit our needs to specify the requirements, these modifications are based on our experiences distilled from several industrial case studies. Our improvements make the formulation of user stories a simplified process. We used Eclipse Modeling Framework (EMF) to describe the metamodels.

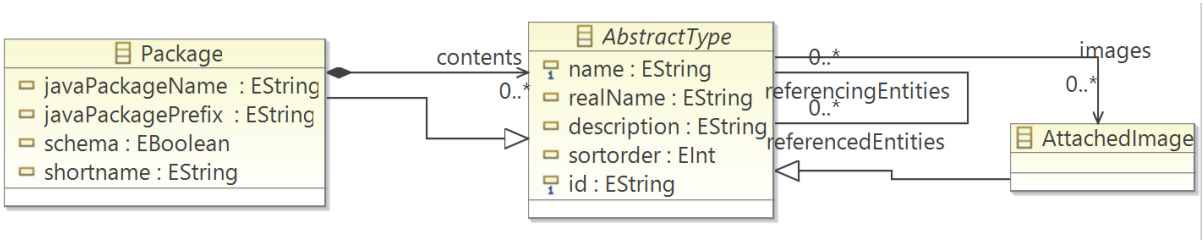


Figure 4-2 Metamodel of the common language elements

The four DSLs are realized as one monolith metamodel because of the requirement to be able to mix the model elements from the different domains into one instance model as well. All languages are based on a common metamodel fragment that is depicted in Figure 4-2. Each element of the metamodel derives from *AbstractType* that facilitates the unified handling of all model elements: each model element has a *name,* a *real name* (to provide a more meaningful name), a unique *id,* a *description*, and a *sortorder* that specifies priority during documentation generation for a specific element. The *description* field contains a rich text description (in XML format) that may contain cross references to other model elements as well. This weak reference (stored in the XML document) is also expressed with modeled references (*referencingEntities* / *referencedEntities* edges) to be able to discover dependencies without parsing all the XML descriptions as well. The descriptions may embed images as well; this dependency is expressed by the *AttachedImage* element (that points to a file-system element) and the *images* relationship.

According to the metamodel, all model elements can be organized into *Package*s that may also correspond to real Java packages during code generation.

4.2.1.2 The Use Case and the Activity DSLs

The *Use Case DSL* is a model of how users connect to and interact with the system to handle problems. The DSL describes the goals of the users, the interactions between the users and the system, and the required behavior of the system to satisfy these goals. In the metamodel of the *Use Case* language (Figure 4-3), the two main elements of the language are the *UseCase* and the *Actor*. Note that all the elements in this metamodel inherit from the *AbstractType* element, so they all have a *name*, a *description*, can be referenced, etc.

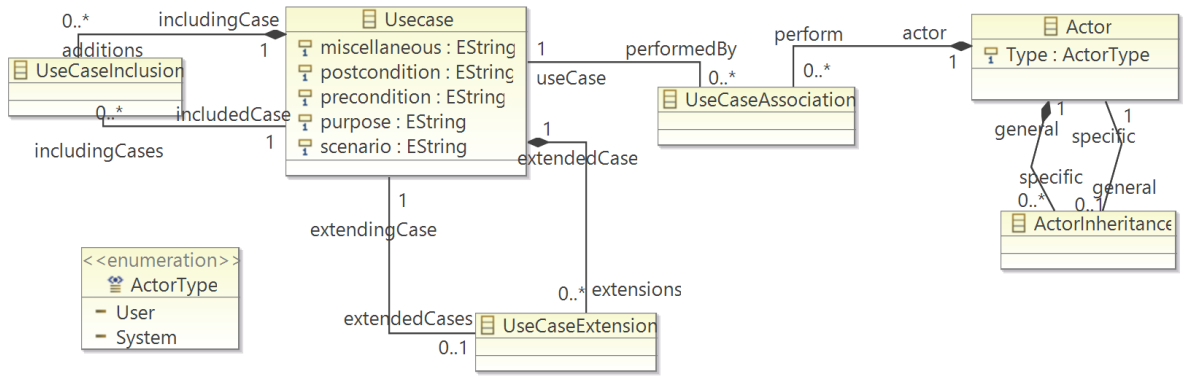


Figure 4-3 The Use Case metamodel

UseCases have the usual *precondition, postcondition, purpose,* and *scenario* attributes to precisely define the use case. The attributes also contain rich text content in XML format, may contain references to images or cross references to other model elements. *UseCases* may include and extend each other corresponding to UML.

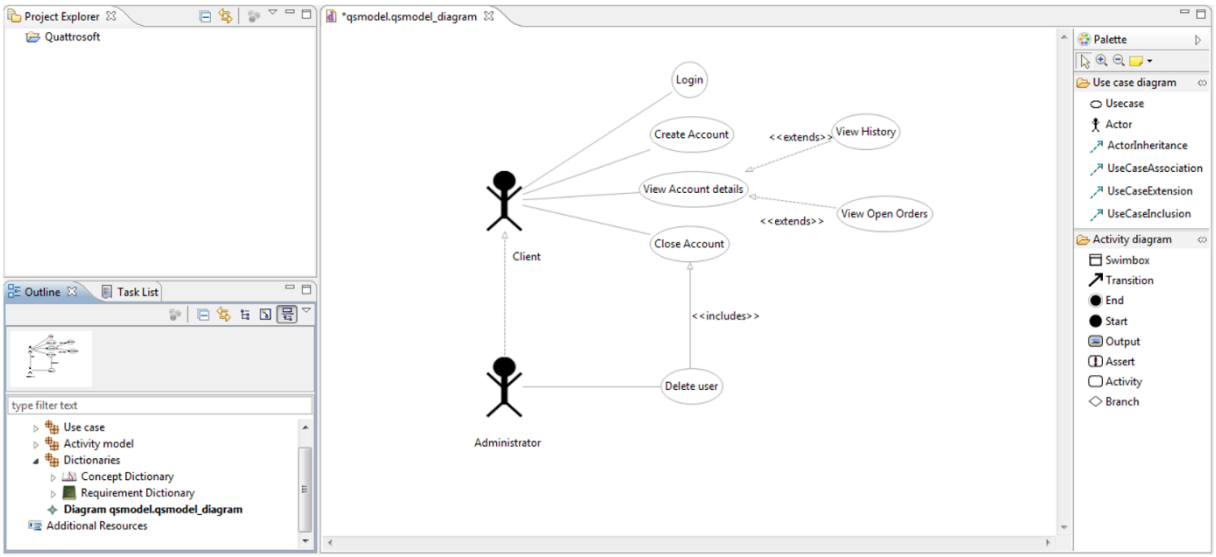


Figure 4-4 A sample Use Case diagram

By *Actors*, we distinguish *User* and *System* actors (*ActorType)* to be able to differentiate between the different actors interacting with the system.

Models can be edited in the usual EMF tree view-based model editor, however, we can assign diagrams to any of the packages as a root element, and visualize a model fragment in a graphical way as well. We may place elements contained by another package than that of the diagram root onto the diagram, however, in this case the element is placed as a *shortcut* (that is indicated by an arrow on the top of the

element). By deleting a shortcut, the element itself is not deleted from the model. In addition, a shortcut is created when a model element is placed on the same diagram for a second or third time, e.g. to increase transparency.

The complete DSL diagram editor application consists of the *Project Explorer* view, an *Outline* view (or *Navigator* window), a drawing area (diagram), and the palettes, which provide the toolset for placing elements on the drawing area (Figure 4-4). Elements can be inserted by clicking on an appropriate element on the diagram palette and dropping it to the drawing view. The navigator window is used as an instant summary of our model as a tree view. We can add, modify and delete new elements here and diagrams as well.

The *Activity DSL* is used to display a sequence of activities. It is useful both to formalize user stories and also to define test scenarios for validation. The metamodel of the language is provided in Figure 4-5. The user story or test workflow starts at a *Start* node, finishes at an *End* node, and each step (activity) describes the test case to perform or the expected behavior in textual, human readable format (*description* field). Activities are connected with *Transition* edges that symbolize a conditional (*guard* attribute) transition between two activities. In addition to simple activities, we distinguish two special kinds of activities as well: *Assert* and *Output*. *Assert* describes the main purpose of the test scenario, i.e. the condition that must be satisfied in order to say that the test has been successful. The *Output* node denotes an activity where the user performing the test must produce some kind of output (a screenshot or a print), and attach it to the test log. These two special kinds of activities are used to emphasize their roles on the user interface for the user and they are also taken into account later, for example, during the automated test generation.

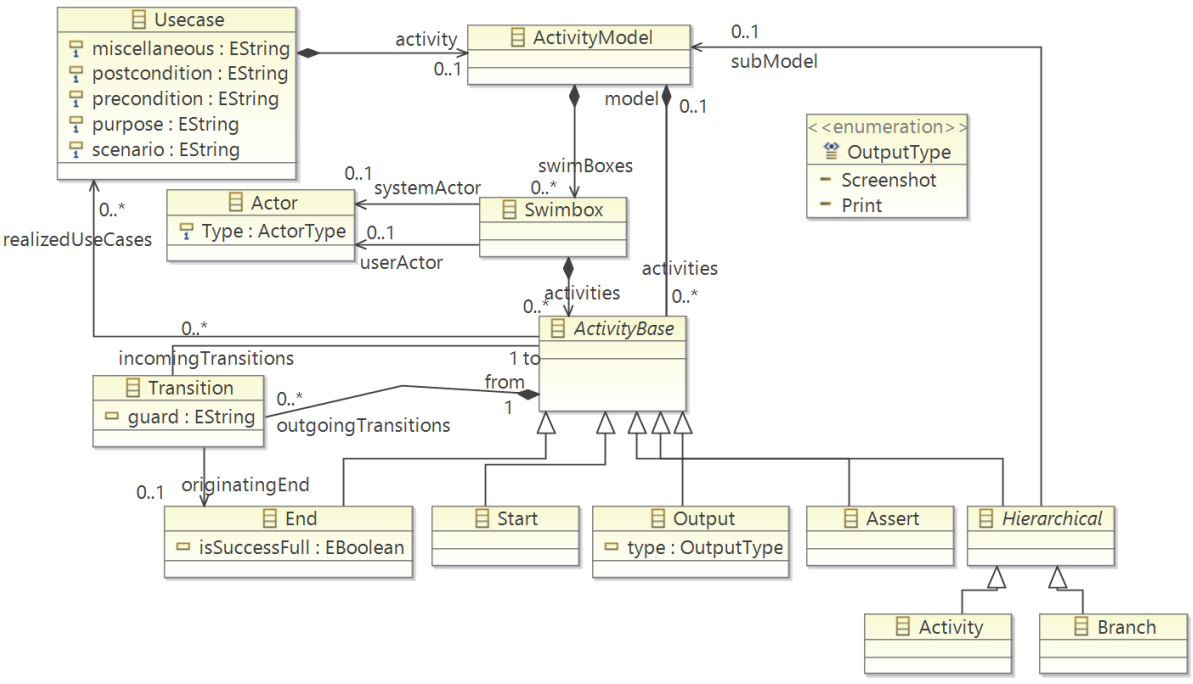


Figure 4-5 The Activity (user story) metamodel

All these techniques, based on the activity model, support the identification of relevant test cases. Activity and use case models represent the business processes and the system level use cases. These models serve as a communication platform at an appropriate level of abstraction. These artifacts can be transformed to human readable and understandable documentations. Therefore, these artifacts are typical products of the requirements handling and definition project phases. Based on the customer

feedbacks, we found that using our DSLs, the model and the generated documentation are compact and unambiguous. The formal definition of the DSLs makes possible to use their keywords as instructions during the analysis of the activity graph.

Activities may be hierarchical as well: an activity can be specified in more details in another activity diagram. The end nodes of the contained activity model can be mapped to the leaving transitions of the container activity node (*originatingEnd* reference), thus, we can specify which transition to follow for each end state of the contained scenario.

To be able to organize activities by actors within a model, we have introduced a special element called *Swimbox*. Swimboxes are similar to swimlanes (like in UML Sequence diagrams) with the difference that they can be freely aligned on the screen thus we can achieve optimal layout. Each swimbox can be connected to one specific user and a system actor.

For an activity, we can also exactly define those use cases realized by the activity (*realizedUseCases* edge), and for each use case, we can assign a complete activity model, if the use case can be performed by a complex workflow. An example activity model is depicted in Figure 4-6.

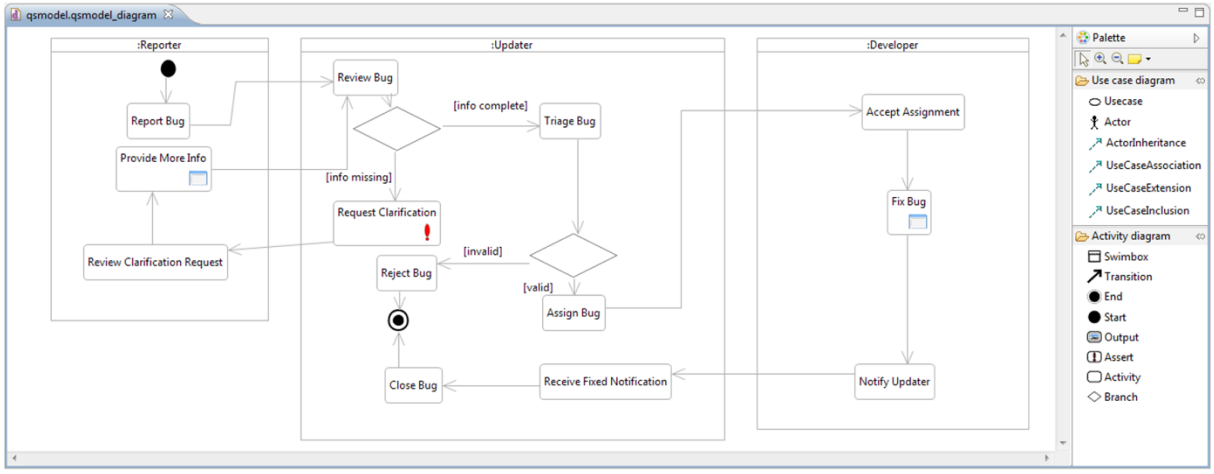


Figure 4-6 A sample Activity diagram

4.2.1.3 The Requirements and Concept Dictionary DSLs

The *Requirements DSL* and the *Concept Dictionary DSL* make it possible to summarize all the related concepts and requirements of an actual software product. The metamodels are provided in Figure 4-7. The implementation provides these dictionaries in a table format.

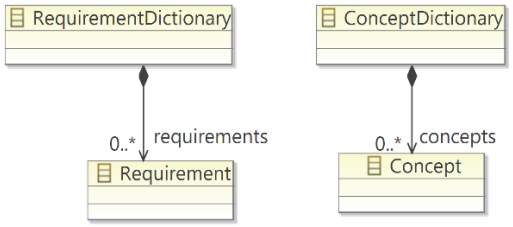


Figure 4-7 The Requirements and the Concept dictionary metamodels

The different dictionaries can be embedded into our model hierarchy. After creating the dictionaries in the tree view of the model editor, we can edit them both in the table view and in the tree view. The table view processes the actually opened model and organizes the related information to display the dictionaries in a readable format in table cells.

The *Concept* dictionary is a glossary of domain related concepts. We can refer to these concepts from other model elements, such as activities or actors, and we can look up those model elements, in which these concepts are affected. An example concept dictionary is presented in Figure 4-8.

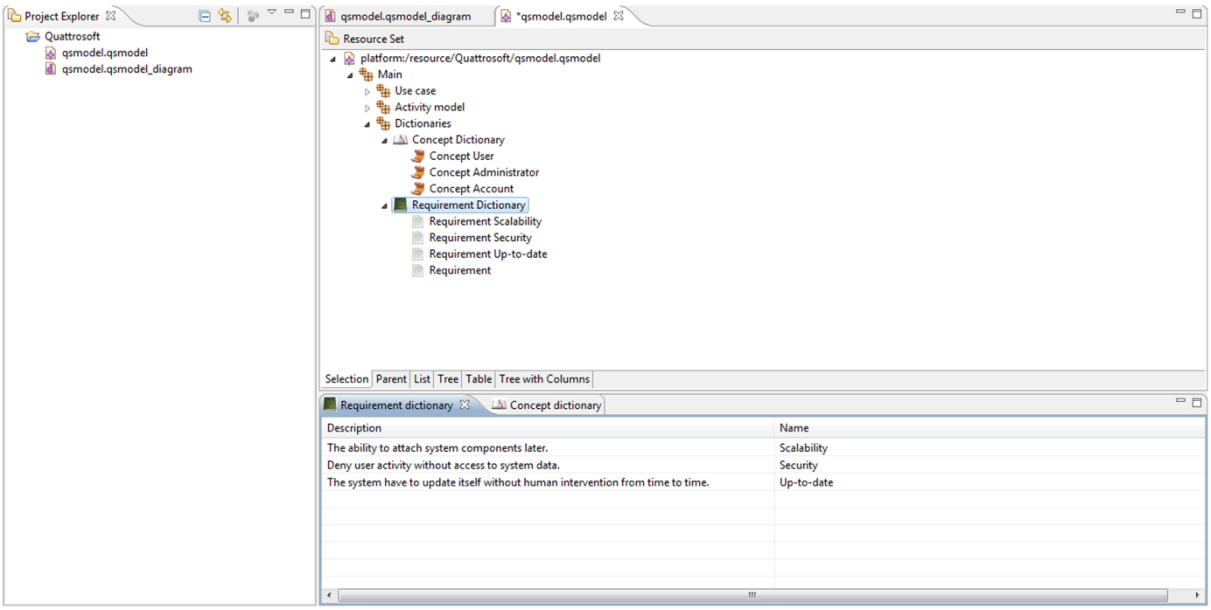


Figure 4-8 A sample Concept dictionary specification

User requirements are collected in the requirements tables. The links between the requirement items and model elements help us to follow which model parts realize a certain requirement, and vice versa, which requirements are affected by a model element. The layout of the requirements dictionary matches that of the concept dictionary.

4.2.1.4 EEF-based Rich Editing Features

The Extended Editing Framework [Eclipse EEF] is a presentation framework for the Eclipse Modeling Framework. EEF makes possible to create rich user interfaces instead of the default grid-like property editor panels to edit EMF models. The realized EEF-based Rich Editor is a great help during the modeling. The capabilities of the realized EEF editor are discussed in this section.

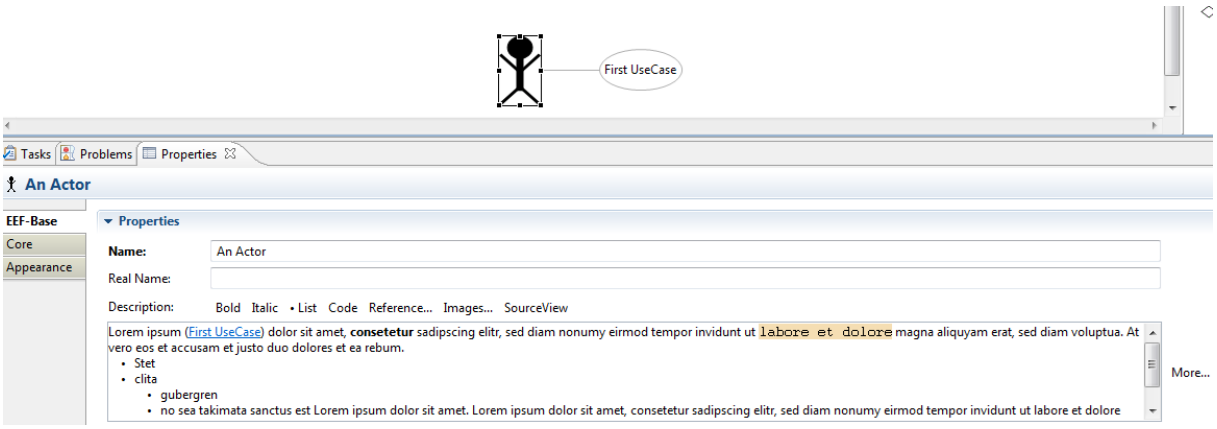


Figure 4-9 The EEF editor of an *Actor* object

In addition to being able to edit the usual model element properties, we can edit the textual model properties with the help of a rich text editor. Using this control, we can apply different font styles (bold, italic) in the text; we can apply bulleted lists, and mark text fragments as source code with special

formatting. References that refer to model elements (e.g. an actor or activity) can be inserted into a text as cross references. Besides, the images provided by the file system (the Eclipse workspace is the root directory) can also be inserted as references (Figure 4-9).

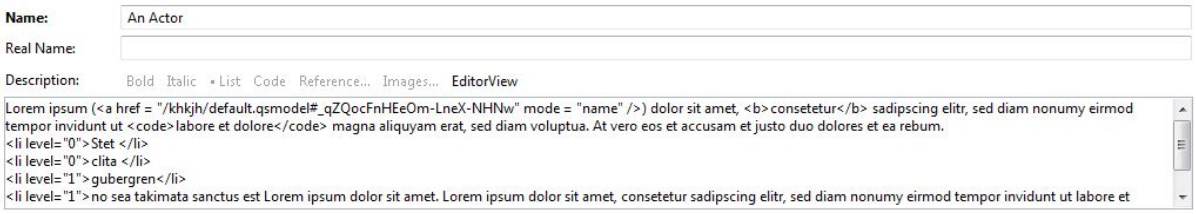


Figure 4-10 The *SourceView* of the Editor

The formatted text is converted and stored in XML format that mostly uses the well-known HTML tags. The XML format of the text is available and editable in the *SourceView* of the editor (Figure 4-10). Pressing the *More*... button, a larger dialog window pops up and makes possible the more comfortable editing of the text.

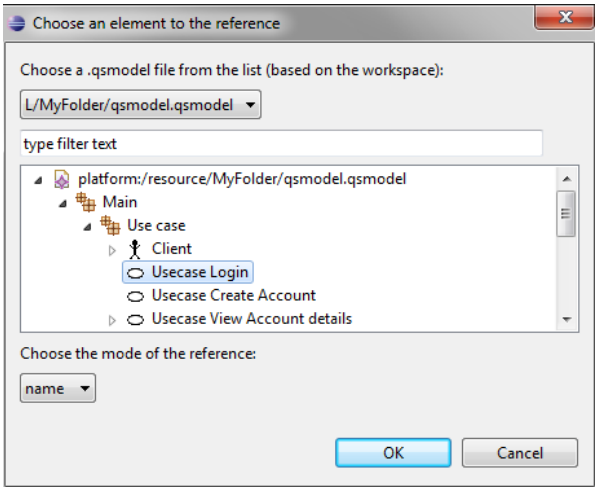


Figure 4-11 The Reference chooser dialog window

A separate dialog has been developed to serve the reference management (Figure 4-11). The window represents the model elements in the hierarchy they are stored in the model. By selecting a model element, the solution puts a reference into the source view of the text (e.g. *<a href = "/MyFolder/qsmodel.qsmodel#_h8XcYDfgEeOAZYngIZUwOw" mode = "name" />*, where the code after the hashmark character denotes the unique id of the referenced model element), and shows the value of the *Name* attribute of the referred model element in the rich text. The name is shown as blue underlined expression, and by moving a mouse over the name of the referred element, a tooltip shows the path related to the model element (Figure 4-12).

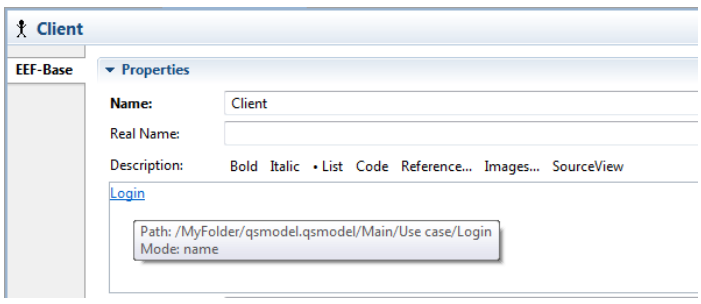


Figure 4-12 Tooltip of a referred element

In a similar way, we can refer to images as shown in Figure 4-13. By selecting an image, the solution puts a reference into the source view of the text (e.g. **), and shows an image placeholder. The tooltip of the placeholder shows the path of the referred image file.

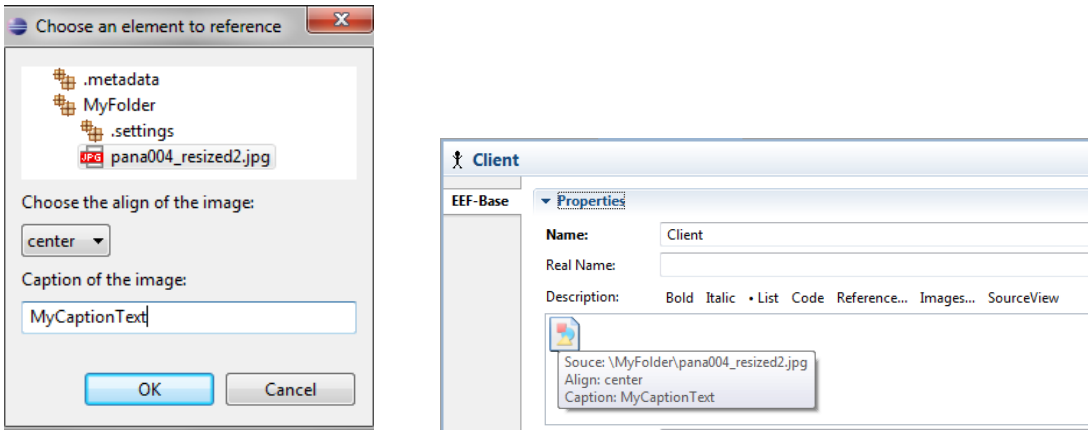


Figure 4-13 The *Image* browser dialog window and a sample inserted image placeholder with a tooltip

With the help of the developed editor interface, we are able to describe use case descriptions, activity operations, etc. in a WYSIWYG format, and we can insert image references and cross references to elements. The presented solution provides a comfortable way to manage cross references that can be resolved during the generation of the documentation, while it also maintains a formal model in the background that facilitates finding dependencies between various requirements and their implications.

4.2.1.5 Further Domains of the Environment

Through a software development process, all the gathered information pieces should be organized in a carefully designed repository structure. Usually the information is represented by plain text documents and stored in separated repositories maintained by the different project partners. This method results in extra administration cost during the software development process. Unfortunately, this type of administration should be performed by highly qualified experts, because all information has its semantic meaning. Misunderstanding the meaning of information can lead to communication problems and further issues. Therefore, applying the appropriate DSLs and providing a consistent working environment for the experts, we can significantly reduce the unnecessary administration and allow experts focusing on their main objectives.

We have defined several further DSLs for the rest of the software development process. All DSLs are about to capture the appropriate level of abstraction and help the communication between domain experts and software engineers. These DSLs support the task management, definition of business entities, user interfaces, etc.

Project tasks are automatically derived from the domain-specific models and provide an easy way to follow the connections between the requirements and the daily activities. The well-formed domain models support automatic code generation. This is important that in this case the code generation is not equivalent with the visual programming technology. Domain models capture a higher level of abstraction than source code. Models support the designer to make high-level decisions. Code generators extend the models with platform-specific and implementation related details.

Business entities of the software systems are defined within a separate DSL. The instance models of this DSL refer to relational database models. We also defined a DSL for capturing the graphical user interface requirements. During the definition of our DSLs, we aimed that the different DSLs can refer

to each other. As a result, different domain models can refer to each other, and they utilize the objects and constructs of each other.

4.2.2 Generating Software Artifacts

Previous section has discussed the requirements engineering part and introduced the supporting environment of our method. Hereby, we summarize the model processors that based on the requirement models generate different artifacts: source code, tasks, documentation and relevant test scenarios.

4.2.2.1 Source Code and Task Generation

The requirement models serve as a plan for all the activities done by different roles of the software development process. They act like a design layout for a building. We found that ideally the great majority of the development related tasks should be generated from the models. The more tasks are generated from the models; the less inconsistency can be observed between the model and the reality. Therefore, the first step is to solve the automatic and tool supported relationship between the model elements and the tasks. The next step is to identify which tasks are well defined and can be expressed as decision points in domain models. For these cases, we can address the generation of the given software artifacts. We believe that this approach is quite important and should be followed; otherwise, a "l'art pour l'art" effort could be taken: generating something that is not relevant or making all the decisions already in the model space.

We apply source code generation to process the following domain models:

- From business object models, we generate the object persistency layer of the system.
- From the abstract user interface models, we generate the appropriate client source code.
- Furthermore, other artifacts are automatically produced like language resources and messages based on the terminology of domain models. It is important to handle this issue in an automatic way, because the same labels, domain expressions and messages should be generated in the documentation, in the source code and on the user interface as well.

4.2.2.2 Generating Relevant Test Scenarios

Based on the business processes and the use case scenarios, the test cases are produced by traversing the process graph and identifying its different paths. The processing generates the so-called *relevant* test scenarios. Relevant test scenarios represent a subset of the complete test scenario set. The solution makes possible to define test data sets that are harmonized with the decision points and the process graph. By exhaustively traversing the whole process graph, we can generate all the test scenarios. However, in case of knowing certain decisions (parameters or variable values) we should test only those branches of the whole process graph that can occur. This makes the testing process more effective, because the known parameters and variable values cut the problem space. For example, consider the following: there is a decision point, where we can choose between the red and green paths. Assume that we select the green one (e.g. based on an input parameter). Later, if the same decision arises, we should follow the same path again. Therefore, there is no need to generate test scenarios covering the red path. In this way, we can reduce the number of test cases.

4.2.2.3 Generating Documentation

In the approach, domain models drive the whole development process. However, end-user artifacts are mainly documents, which are the readable layouts of the given domain models. The business process models are a communication platform between the development team and the project stakeholders. Therefore, the documentation generated based on these models should be well-formed, and furthermore, easy to read and understand for the domain experts. To fulfill this requirement, the business process

model and every further DSL sections have their own documentation layout. The generator uses a template-based technology by writing the document directly through an API. The generator can produce documents of several hundred pages within a few seconds, i.e. the document generation solution is effective. Note, that it is not a goal to generate verbose and too long documents. The length of the documents depends on the sections related documentation layout, the size of the processed domain models, and the document format settings (e.g. font size, margins and further paragraph settings).

Model element descriptions are formatted as rich text sections; furthermore, they handle inserted images as well. Therefore, the generated document artifacts do not require any post-processing like manual formatting or extending.

4.2.3 Evaluation of the Method

The result of this systematic approach leads to a close relationship between the original requirements, the source code, the documentation, and the test cases used in the transition phase of the project. Based on our experience, the method significantly reduces the project risks. The method is a strong tool for supporting change management.

Generally, both parties (the customer and the solution provider) are interested in defining the system requirements as precisely as possible. Our projects proved that extra efforts (e.g. a careful design) performed at the preliminary phase save more efforts in the transition phase and significantly reduce the risks. This was the main goal during developing and using this method.

The strengths of the method:

- Requirements models drive the whole development process: requirements analysis and requirements engineering, artifact generation (development related tasks, source code, test scenarios, and documentation), testing, verification/validation issues, maintenance, and feedback management. This results a model-centered and strictly model-driven approach with advantages of clear requirements management, direct connections and consequences during the development process.
- Provides the frame of adequate analysis, unambiguous scope definition and involvement of the domain experts (customers) into the development.
- Defines a clear connection between the requirements and the software components (system functionalities).
- Puts the domain processes (workflows) into the focus of the requirements analysis and the development.
- Ensures the continuously up-to-date state of the requirement models all along the lifecycle of the given solution.
- The suggested method can be freely extended with optional number of DSLs and domain-specific model processors. The key concept is that models should drive the artifact generation, the testing process and also the maintenance.

The weaknesses of the approach:

- Each research team and developer company has its own rules and processes. The introduced method is not about to redefine the working method, i.e. the method is not about to take and use it as it is, but take and adapt it. We suggest thoughts and assets that are worth to follow in order to make the software development process more effective and increase the quality of the resulted artifacts.

- The generated source code is based on organization-specific class libraries or frameworks. Some of these libraries are developed for different application domains (e.g. financial sector, governmental sector, pharmaceutical field, other) but the great majority of these libraries cover enterprise software related issues, i.e., support the communication between the clients and the server, the data access and database management issues. These libraries are team-related artifacts, therefore, the generators should be redeveloped (or heavily reconfigured) by each research team or company in order to target their own library capabilities.

The introduced method has various benefits for both the academy and the industry. By applying quality assured methods in requirements analysis, user story definition, and requiring customer validation of the business requirements, guarantees the unambiguous project scope definition, the appropriate quality assurance, and the customer satisfaction. We believe that this method can significantly improve the quality of the software artifacts, increase the development productivity, and decrease the required resources and the time-to-market period. The methods and the introduced domain-specific languages are tool independent, i.e. the results and the methods can be applied in different environments as well.

4.3 Developing and Managing Domain-Specific Models

By *domain-specific languages,* we mean textual or visual languages that are used in a more specialized way than in general-purpose programming. DSLs have limited expressive potential and can only describe problems from a well-defined problem domain. These characteristics make them suitable to achieve several different intents.

Domain-specific languages (DSLs) have been used for a long time in the software world as a means for achieving better productivity and increasing the quality in final products. DSLs are used extensively, so when we talk about productivity, it may refer to the efficiency of creating both physical products and intellectual property, depending on the production environment. Some DSLs are destined to end-users. In this case, they facilitate the use of a software product so that the end-users can work more efficiently.

Domain-specific languages play an important role in software modeling and model-driven methods. A number of international journals and conferences discuss the topics of the professional development of languages, their exploitation, furthermore the benefits and opportunities provided by their application. Domain-specific languages are utilized in various scenarios, such as a tool specialized for a domain; as a customized notation; as a modeling tool; as a software layer; or as addressing the challenges from a different point of view. Development, implementation and utilization of domain-specific languages requires mature decisions, analysis, design, implementation, introduction and maintenance [32].

We have worked out and tested a method with the goal to allow the effective, reason and utilization-driven definition of domain-specific models. The method provides suggestions regarding to the considerations and decisions we have to make during the analysis of the business needs and the definition of domain-specific languages, supports the steps and tasks related to the introduction of domain-specific languages. The method also provides suggestions regarding to the maintenance of domain-specific languages.

Introducing a DSL is an investment, which should pay off. The characteristics of DSLs are much more than just the limited expressive potential, i.e. focused intent, and higher abstraction view.

Usually, DSLs are defined as computer languages that use domain-specific notions to describe a computer program. Designing a DSL also a kind of development that requires goals and scope. Before we introduce a DSL, we need a clear vision, i.e. the motivating points to use a DSL, which aspects are in focus and what are the goals to be achieved by their application. Furthermore, we should be aware of the implications of the DSL implementation, the advantages and disadvantages, the challenges that we

face during the design, development, introduction, utilization and maintenance of the DSL. Often, we handle a DSL as a *specialized tool* with a *custom notation* for a given area, for a given type of development activities [32].

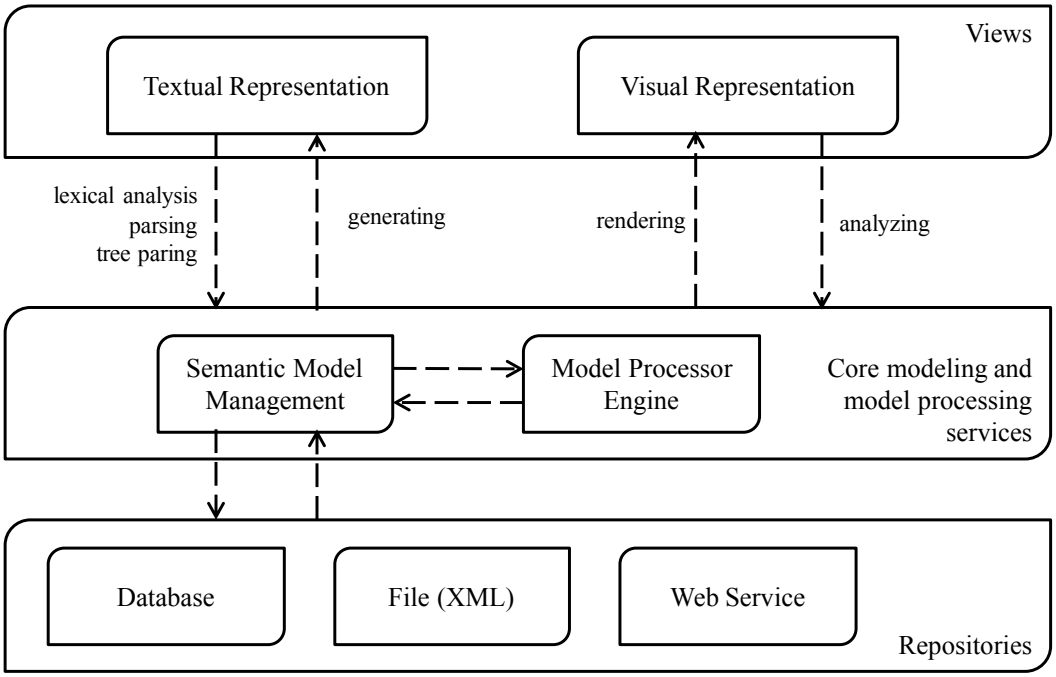


Figure 4-14 Supporting the transparent switch between the textual and visual views of semantic models

Figure 4-14 depicts the architecture of our method for managing domain-specific models, which supports the transparent switch between the textual and visual views of the semantic models. In the architecture, there are three layers: the repository layer; modeling and model processing services; and views. The *repository* media can be a classical file or database storage, or even any communication point, such as a web service, which can accept domain models and transparently provides them on request. The *core modeling and model processing service* represents the semantics of the model, furthermore, this layer is responsible for the model processing activities. Model processing engine, validated model processing methods, code generators are all represented by this layer. A *view* is based on the semantic model and provides optional textual and/or graphical representation for software models. The approach is similar to the Model-View-Controller and the Document-View design patterns. The various responsibilities of the modeling and model processing field are clearly separated and transparently applied in order to provide an effective method for large-scale software projects.

The method allows the transparent switch between the visual and textual representations of semantic models. Furthermore, it allows to freely decide, whether to use the textual and/or the visual representation of domain-specific languages to define the structure or the operation of software systems.

4.4 Processing Mathworks Simulink Models with Graph Rewriting-Based Model Transformations

Simulink, developed by Mathworks, is a graphical programming environment for modeling, simulating and analyzing multi-domain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink is widely used in automatic control and digital signal processing for multi-domain simulation and model-based design. [Simulink]

Simulink well demonstrates that increasing the abstraction level is a powerful tool to manage complexity. Developer teams appreciate that model processors targeting embedded environments, controlling systems, or signal processing, are also enable to generate software artifacts, e.g. configuration files, execution lists. Simulink does not support the declarative definition of model processors. Therefore, we have prepared VMTS modeling and model-processing framework to support the communication with the Simulink environment. We define it as our integration between the Simulink environment and the VMTS modeling and model processing framework. VMTS allows to define and execute model processors. As a result, tasks are designed and solved on their appropriate abstraction level, i.e., we traverse, optimize and alter models in the Simulink domain. The Simulink metamodel has been automatically built in the VMTS environment, therefore, engineers use well-known Simulink language elements during the design of model processors. This is the fundamental premise of Computer Automated Multi-Paradigm Modeling; to use the most appropriate formalism for representing a problem at the most appropriate level of abstraction [Mosterman and Vangheluwe, 2000].

The integration allows to load Simulink models and to perform the model processing rule-by-rule, while continuously modifying the Simulink model and/or generating the output of the transformation [25] [30].

While operating at a given level of abstraction, two mechanisms are often employed to scale system complexity: partitioning and hierarchy [Mosterman et al, 2004]. In the Simulink environment, hierarchy is supported as a purely syntactic construct by virtual subsystems and as a construct with semantic implications by non-virtual subsystems. These subsystems are represented as blocks with input and output ports that are used to connect subsystem blocks. Subsystem blocks may contain other subsystem blocks or primitive blocks that represent behavior without being able to be further decomposed.

Before a Simulink model is executed, the engine creates an execution list with an order in which all of the blocks are executed. The execution list is computed from the sorted list, which is also generated by the Simulink engine based on the control and data dependencies that determine how the different blocks can follow each other in an overall execution. To create this list, the semantically superfluous hierarchical layers have to be flattened. So, the virtual subsystems that are only graphical syntax and that have no bearing on execution semantics are flattened before the sorted list is generated.

We have applied the graph rewriting-based solution to flatten virtual subsystems in Simulink models. Model transformations helped to raise the abstraction level of the transformation from the API programming to the level of software modeling. The solution possesses all the advantageous characteristic of the model transformation, for example, it is reusable, transparent, and platform independent.

4.4.1 Communication between Simulink and VMTS

Since models are defined in Simulink, which is a part of the MATLAB environment and the transformation is defined in a different system (VMTS) there was a need to establish a communication method between the two systems.

To be able to represent Simulink models in VMTS, the metamodel of the Simulink languages, which are organized into different libraries (also called *blocksets*), is required. In Simulink, there is no hard boundary between the different languages, that is, a given block can be connected to almost everything else, a common Simulink metamodel was created. The metamodel contains all the elements of the Simulink library. The generation of this metamodel consists of the following two steps.

First, a *core* metamodel was created that contains the *Block* element, which is the common ancestor of all the nodes in Simulink models, and a descendant *Subsystem* node, which expresses the common

ancestor of Simulink Subsystems. This metamodel also contains the *Signal* edge and a *Containment* edge to reflect containment hierarchy between nodes.

Then, by programmatically traversing the base Simulink library, this metamodel has been extended with the other nodes found in the different specialized libraries. For each Simulink element, exactly one node was generated. This resulted in several hundred new metamodel elements.

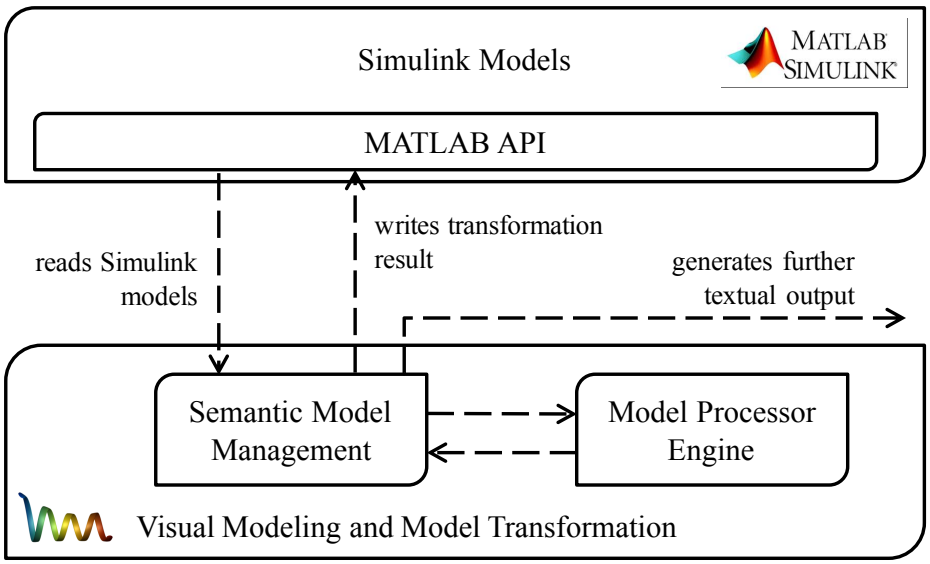


Figure 4-15 Processing Mathworks Simulink models with graph rewriting-based model transformations (within the VMTS Framework)

In addition to the metamodel, the VMTS was prepared to read and write Simulink models. Thus, a new kind of data exchange layer was generated for communicating with MATLAB. To modify Simulink models, the P/Invoke technology [P/Invoke] has been chosen. This has the advantage that the MATLAB interpreter can be called directly through DLL calls, instead of manipulating the textual model (mdl extension) files. This way the VMTS is independent of file format changes, and the changes performed on the VMTS model can be made visible, live during the transformation execution, on the Simulink diagrams as well. Furthermore, the values that are only available during simulation time of a Simulink model can be accessed also.

Figure 4-15 introduces the architecture of the Simulink and VMTS integration, i.e. the method for processing MATLAB Simulink models with graph rewriting-based model transformations.

4.4.2 Visual Debugging Support for Graph Rewriting-based Model Transformations

Within VMTS, we have developed an interactive visual model transformation debugger, i.e. a visual debugger for graph rewriting-based model transformations [31]. The motivation was mainly to support the effective development of the MATLAB Simulink model processors.

Actual model processing tools support the definition of transformation rules in a visual or textual way. Several tools support both forms. When developing a model processor for a domain-specific language, the ability to efficiently trace and debug the operations performed by the model processor is also important. Almost all software development tools facilitate the debugging of source code during the execution. But only a few of the modeling tools support the continuous animation of the modifications performed on the models, which makes the traceability and the debugging of transformations challenging.

We have worked out the concept of the visual model transformation debugger and realized it in the VMTS. The solution facilitates the step-by-step execution of model transformations, breakpoints, variable inspection, and the visualization of the overall state of the transformation. Furthermore, it provides the possibility to influence the behavior of the transformation at runtime. The realized features have successfully applied for the MATLAB Simulink and further domains.

4.5 Managing Energy Efficiency-related Properties

Energy efficiency is a critical attribute of mobile applications, but it is often difficult for the developers to optimize the energy consumption on the code level. In this chapter, we explore how we could use a model and code library based approach to assist the development. Our vision is that engineers can specify the operation on a high level and the system automatically converts the model to an appropriate software pattern. In this way, the developer can focus on the actual functionality of the application.

Research has shown that by developing applications in a smart way, major savings in energy consumption can be achieved. For instance, [Hoque et al, 2013] show that up to 65% of the energy consumed by wireless communication can be saved through traffic shaping when streaming video to a mobile device, and [Nurminen and Noyranen, 2009] report 80% savings when data transfer happens during voice calls. These results show that when an application is properly constructed it is possible to fundamentally improve the energy efficiency of at least some classes of mobile applications.

Unlike energy efficiency improvements in hardware or low-level software, these application-level techniques require extra development work. Increasing the awareness of energy efficient programming techniques could be one way but educating the vast mass of mobile developers to know the needed patterns is slower.

Increasing the abstraction level is a successful way to hide low-level details from the developers, which makes it possible to use complex data structures without worrying too much how they are actually implemented, or we can use a high-level communication protocol without worrying how the bits are actually moving through the communication channel.

The key question we investigate is the following: How can we hide some of the complexity needed for energy efficient operation behind a properly selected abstraction? Some related work exists that touches the interface between applications and power management of mobile devices: [Anand et al, 2004] propose specific kind of power management interfaces for I/O access that allow applications to query the power state and propose middleware for adaptive disk cache management. In [Anand et al, 2005], the authors propose to leverage application hints in power management. However, choosing the right level of abstraction in order to make the energy efficient programming techniques and patterns widely and easily available to the developers still seems to be an open issue and requires further exploration. As far as we know, our approach was the first one to investigate how the techniques and coding-patterns needed for energy efficient software could be made widely and easily available to the developers.

An ideal solution would also enable distributing newly discovered patterns and techniques later on to the software developer community without requiring further efforts from the developers.

In summary, we propose a way to reduce the complexity of developing energy efficient mobile applications. The key contributions are:

- We identify energy efficient software patterns that can reduce the cost of communication.
- We propose a way to hide most of the complexity of the patterns behind a high-level software abstractions and discuss what kind of issues arise with such an approach and what are the remaining open problems.

- We specifically advocate a model-driven approach that is one of the possibilities to address our goals, i.e. to effectively support the development of energy efficient mobile applications.
- In the discussion section, we collect the main questions and issues of the field and provide our thoughts about them.

In [35], we provided all the details of these results. Here we focus on the modeling and model processing aspects of the method.

We have identified and analyzed the following energy efficient communication patterns, that can achieve smaller energy consumption compared to the unregulated transfer of data:

- ***Delayed Transfer***. Data transfer requests are queued and transmissions happen in a batch after a certain time has passed or the size of the data to be transferred is large enough. The energy saving in this case is the result of the fewer bursts of data which means less energy overhead from the switching of radio states.
- ***Bursty Transfer***. Faster communication incurs lower energy per bit cost with radio communication [Nurminen, 2010]. The main idea is that the two communicating partner reschedules the transfer session in a way that the data is transferred in high rate bursts instead of at a steady but slower rate. This differs from the delayed transfer pattern, where the complete data payload is always sent in one burst.
- ***Compressed Transfer***. The most generic solution to save energy [Barr and Asanovic, 2006]. It involves compressing the data to be transferred at the sending device and decompressing it at the recipient after the transfer session.

4.5.1 Modeling and Generating Energy Efficient Applications

In order to support the energy efficient patterns, we designed an architecture and implemented a prototype system for Android platform. Based on the model, which describes the energy aspects of the application, the generated code uses a library, which interacts with our runtime system. The runtime system, also referred to as Energy Efficient Communication Service, implements various energy efficient patterns and takes care of the related tasks.

Modeling and processing different aspects of mobile applications are performed in the VMTS framework. We use mobile-specific languages to define the required structure and application logic. Platform-specific model processors are applied to generate the executable artifacts for different target mobile platforms. The generated code is based upon the previously assembled mobile-specific libraries. These libraries provide energy efficient solutions for mobile applications through the runtime component. (Figure 4-16)

Mobile-specific languages address the connection points and commonalities of the most popular mobile platforms. These commonalities are the basis of further modeling and code generation methods. The main areas, covered by these domain-specific languages, are the static structure, business logic (dynamic behavior), database structure and communication protocol. Using these textual and visual languages, we are able to define several parts of the mobile applications.

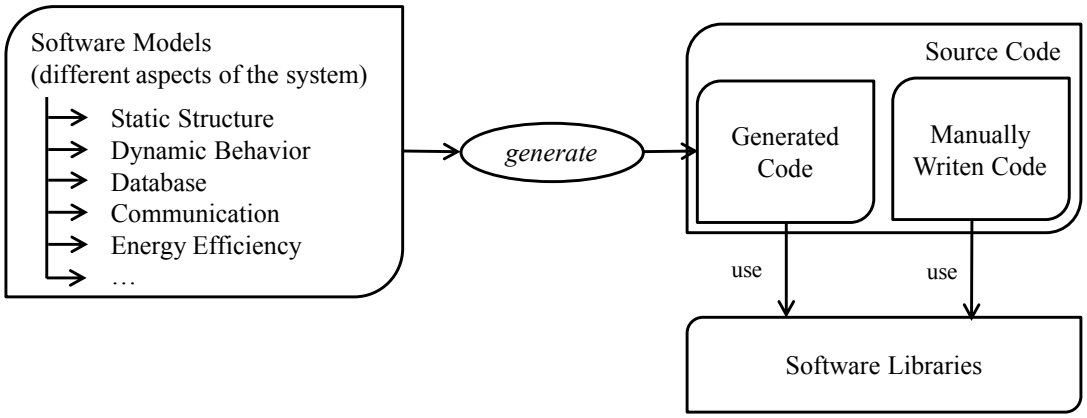


Figure 4-16 Managing different aspects, including the energy efficient operating properties, of software systems on the modeling level

The following example introduces a textual language-based sample for defining REST-based (Representational State Transfer) communication-related interface. The solution can generate both the server and client-side proxy classes supporting the communication based on these type of interface definitions.

```
[RestApi]
public interface MyService
{
  [RestMethod(Url = "$client/insertuser.php", CommandType
    = CommandType.POST, ReturnFormat = FormatType.XML)]
  UserInfo InsertUser ([RestParam(Mapping = RestMethodMappingType.Custom)]string client,
    [RestParam(Name="usr", Mapping = RestMethodMappingType.Body)]string userName);
}
```

For each target platform, a separate transformation is realized since, at this step, we convert the platform-independent models into platform-specific executable code. The transformation expects the existence of the mobile frameworks and utilizes their methods. The generated source code is essentially a list of parameterized activities (commands), that are certain functions of the mobile application. This means that the core realization of the functions is not generated but utilized from mobile-platform specific frameworks, i.e., the generated code contains the correct function calls in an adequate order, and with appropriate parameters. The advantages of this solution are the followings:

- The software designer has easier task. The model processors are simpler. The model processing is quicker. The generated source code is shorter and easier to read and understand.
- We use prepared mobile-specific libraries. These libraries are developed by senior engineers of the actual mobile platform and subject. These solutions, applying the appropriate patterns, take into account the questions of energy efficiency as well.

In order to handle the energy related aspects of mobile applications also on the modeling level, we should introduce those attributes that enables to define the envisioned conditions and circumstances of the data transfer. This include the amount of the data, the frequency of data transfer, and the tolerated delay.

- The above example is the textual view of the model, of course the modeling framework makes possible to provide a visual interface to edit these and other domain-specific models as well.

There are various solutions that can be utilized for code generation. We have chosen the Microsoft T4 (Text Template Transformation Toolkit) [Microsoft T4]. T4 is a mixture of static texts and procedural

code: the static text is simply printed into the output while the procedural code is executed and it may result in additional texts to be printed into the output.

4.5.2 Discussion

The first of the aspects to be considered is where to place the logic that chooses and configures the patterns to apply. One option is to integrate it into the client side library, which monitoring the current device environment could then operate adaptively. The other option is that the pattern-related decisions are made during the code generation phase by the model processor. As usual, there are pros and cons for both approaches. Having the logic in the client side library is beneficial because it can be configured adaptively. On the other hand, the developer may provide a hint for the library and in this way, the result can be a clearer and more understandable source code.

The next questions are the followings: Why is it important to increase the abstraction level in mobile application development? What are the benefits of modeling mobile applications? Why is it important to include the modeling of energy efficient issues of mobile applications as well? To raise the level of abstraction in model-driven development, both the modeling language and the model processor (generator) need to be domain-specific. The approach can improve both the performance of the development and the quality of the final products. The commonly identified benefits of the model-driven approach are improved productivity, better product quality, hidden complexity, and reuse of expertise.

Moving certain parts of mobile application definition to the modeling level, including the energy efficient settings, means that these aspects of the applications are also available on higher level. This makes it possible to gain an overview of the applications, to analyze them from different points of view, to verify or validate certain critical issues, and to generate the source code of the application from models.

We believe that it is necessary to involve the software developers in making applications more energy efficient at least to a certain extent. Solutions that delay traffic and are completely transparent to the developer, like the one described in [Qualcomm, 2012] may not be optimal either because they might lead to unexpected behavior from the developer's viewpoint. From the perspective of a typical developer, the model-driven approach may sound unnecessarily cumbersome.

However, from the perspective of the product owner, the project manager, the software architect, and the customer it is worthwhile to move all possible aspects of mobile applications to a higher level of abstraction because that is the level at which they typically operate. They cannot think on the level of the libraries (source code), because they are responsible for a whole system with different parts: server side, desktop client, web clients, mobile clients, cloud-related things and so on. Still, they want to see the important aspects of each area. Modeling should concentrate on all key aspects of the mobile applications, including energy efficiency.

Energy efficiency is typically context dependent. This applies especially for data communication where the amount of Joules per bit spent depends on at least SNR (transmit power) and available bandwidth that vary with location and time. Therefore, regardless of whether model-driven or purely library-based approach is used, some logic should be executed at run time. For example, the energy utility of data transfer (J/bit) depends on e.g. SNR and available bandwidth that are location and time dependent which in turn impact the decision whether compressing specific type of content yields energy savings or not [Xiao et al, 2010].

Our examples focus only on communication energy. Other sources of energy drain, such as CPU, display, sensors, can be included to consideration as well. For example, the developer could decide to execute a function that renders the display in the most energy efficient manner by adapting the colors

[Dong and Zhong, 2011]. As for sensors, the programmer could explicitly set the trade-off between accuracy and energy consumption.

In summary, we can say that applying model-driven solutions in mobile application development is the current trend, and in the case of energy efficiency, it is also advised to follow it. Obviously, there are scenarios where modeling (higher abstraction level) does not provide immediate advantages. Our work in this area was the starting point of a discussion within the research community concerning how to effectively transfer the existing and novel energy efficiency techniques to the mobile application development.

4.6 Conclusions

Model transformation approaches are becoming increasingly valuable in software development processes because of the ability to capture and apply domain knowledge on the appropriate abstraction level and often in a declarative manner. This enables various steps in the software development to be specified separate from one another with apparent advantages such as reuse. In system design, the computational functionality moves through a series of design stages where different software representations are used. For example, before generating the code that is to run on the final device or platform, code may be generated that includes additional monitoring functionality.

As a design is being refined, models generally pass through a series of stages where details are increasingly added as a form of model elaboration. Model elaboration can, at least in part, be automated by sophisticated software tools.

A method has been worked out that supports the quality assurance of software development projects. The method applies domain-specific languages to support the compilation and analyzes of business requirements against the software.

We have developed a method that allows the definition and management of domain-specific models. The method provides suggestions regarding to the considerations and decisions we have to make during the analysis of the business needs and the definition of domain-specific languages, supports the steps and tasks related to the introduction of domain-specific languages and finally, helps during the maintenance of domain-specific languages. Furthermore, a method has been worked out that allows the effective and transparent switch between the visual and textual representations of semantic models.

We have successfully worked out and applied a novel solution that raises the abstraction level of processing Simulink models. This approach utilizes model-transformations defined in VMTS. Using model transformation to reduce the risk of creating artificial algebraic loops, replace built-in solutions with custom algorithms, and change data types helps raising the abstraction level from API programming to software modeling. The solution possesses all the advantageous characteristics of the model transformation, for example, it is reusable, transparent, and platform independent.

The transformations were implemented in the Visual Modeling and Transformation System (VMTS). In order to transform Simulink models directly, the VMTS modeling framework communicates with the Simulink environment and the technology for doing so was briefly introduced.

Finally, we have provided a method for defining and managing energy efficient operation properties of mobile devices. The method allows defining the energy efficiency properties on the level of software models. As a result of the solution, this aspect of software systems appears on the modeling level, which provides a more detailed view about the whole system.

The discussed area related selected scientific results from my research activities are summarized in Section 7.2 (Thesis II: Model-Driven Methods Based on Domain-Specific Languages and Model Processors).

I believe that my results, domain-specific languages and model processors driven methods, contribute to the quality improvement of the software artifacts and increase the development productivity. Both the discussed methods and the introduced domain-specific languages are tool independent, therefore, these results can be utilized in various tools and for different circumstances.

5 Applying Domain-Specific Design Patterns and Validating Domain-Specific Properties

5.1 Introduction

In the field of graphical languages, we use metamodeling as a widely applied technique to define both modeling languages and highly configurable environments. These languages and environments support the rapid development of domain-specific languages and model processors. Design patterns are efficient solutions for recurring problems [Gamma et al, 1995]. With the popularity of domain-specific languages and domain-specific model processors, there is a need for domain-specific design patterns to offer solutions to problems reoccurring in different domains. This chapter introduces both theoretical results and practical methods to support domain-specific model patterns, validating domain-specific properties of software models and handling the validating constraints in a modular way.

5.2 Domain-Specific Design Patterns

In [2] we discussed both the theoretical and practical foundations with the goal to support domain-specific model patterns in metamodeling environments. In order to support the treatment of early model parts, we weakened the instantiation relationship. We worked out various constructs that help relaxing the instantiation rules, and showed that the utilization of these constructs allows to express domain-specific patterns.

Metamodeling is a successful technique to define the rules of graphical modeling languages. The method allows to define the metamodel itself and also provides an instantiation relationship. We say that a model is valid if it can be obtained from the metamodel via the agreed instantiation relationships.

Since metamodels allow model descriptions, various language-related operations could be handled in a more general way, if the metamodel is also present with the model. This leads us to the concept of reuse, i.e. an operation assumes a metamodel description language, the instantiation method is the only fixed element, the model and its metamodel become variable parameters, and in this way the features of a modeling tool can be reused across any model in a metamodeling environment.

Domain-specific design patterns are design patterns inserted into domain-specific languages. They are addressing a specific domain-related challenges [11]. In domain-specific languages, not only design patterns can be developed, but model patterns for many other purposes. The modeling language patterns in this broader sense are referred to as domain-specific model patterns. We have developed the method and also the tool support for domain-specific model patterns. The features for pattern specification are general, i.e. they exploit the benefits of metamodeling described above: the metamodels can be taken as parameters in metamodeling environments. However, we need to weaken the instantiation rules to be able to store premature model parts in a metamodeling environment.

In software modeling and graph rewriting-based model transformation, there are several recurring problems that should be solved repeatedly in the context of different environments or different transformations. A pattern is a reusable entity, which describes a frequent design or implementation problem, and gives a general, but customizable solution for it.

Model transformations are formed using transformation rules. Within a model transformation, it is required that each rule be defined over an attributed type graph. This type graph is obtained by regarding the metamodels of both the source and the target languages. As a consequence, one requirement for rules is that both LHS and RHS are instances of their corresponding metamodels.

Often, this instance relationship is loosely applied: we require that a model or a transformation rule must be an instance of the metamodel, but should not necessarily conform to additional constraints (e.g. attribute value constraints). This does not affect cardinality constraints. The reasons of this method are that we do not want to define unnecessary restrictions when designing a model transformation and we aim to reduce the number of rules by allowing abstract classes.

Figure 5-1 illustrates that metamodel-based transformation rules are on the same level as metamodels, furthermore, introduces the instantiation hierarchy applied in a metamodeling tool. The left-hand side of the transformation rules is built based on the types and associations of the input metamodel. Similarly, right-hand side of the transformation rules is built based on the types and associations of the output metamodel. The figure summarizes the instantiation relationships between the meta-metamodel, metamodels, instance models and transformation rules. Domain-specific design patterns are also valid or relaxed instances of the metamodels. We apply and reuse these model patterns with the help of transformation rules, i.e. transformation rules insert the model patterns into the appropriate parts of the instance models.

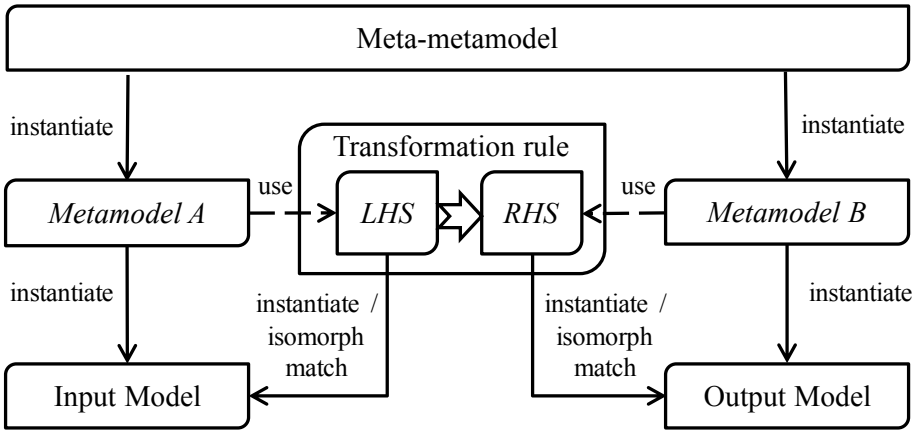


Figure 5-1 Supporting domain-specific design patterns

In [2], the concepts of *partial instantiation, relaxed multiplicity*, and *relaxed instantiation relationship* are introduced. In order to support the treatment of premature model parts, such as transformation rules or model patterns, this approach also weakened the instantiation relationship.

In requiring valid intermediate models after each rule execution, the following question arises: Can we use the same metamodel both for intermediate models and for the final output model? To support the model transformation process with the same metamodel, certain relaxations should be allowed on the instantiation relationship, i.e., special instantiation rules are necessary. In order to describe these special instantiation rules, we need to introduce the concept of partial instantiation.

Definition (Partial instantiation). If a model can be extended to a valid instance model, it is a partial instance of a metamodel. During the extension, it is not allowed to delete or modify any element or attribute value of the partial instance model, but only extension functions, i.e., adding new elements and initializing (uninitialized) attribute values is permitted.

In the majority of the practical cases, partial instantiation means relaxing the multiplicities on the edges and the cardinalities of the nodes in the metamodel.

Assuming that a multiplicity interval is *m1..m2*, where *m2* can be infinity. The relaxed version of this multiplicity interval is *0..m2*. Similarly, assuming that a cardinality interval *c1..c2*, where *c2* can be infinity. The relaxed version of this cardinality interval is *0..c2*.

With relaxing the multiplicities and cardinalities, we facilitate the omissions of certain nodes and edges. However, the patterns in certain cases do not conform to the relaxed model. Assume the following containment chain: *Queue-Task-Subtask*. When we define patterns, it would be desirable to model a subchain of these chains. In general, this means that we allow transitive containment in the model.

Definition (Transitive containment). Assume a directed containment path between two nodes N_1 and N_2 in a metamodel. Transitive containment means that objects of N_1 can contain objects of N_2.

Assume that the modeler has constructed a pattern with the containment chain *Queue-Subtask*. In the editor (at the concrete syntax level), it seems that the modeler must insert a *Task* between the *Queue* and the *Subtask* to extend the pattern to a valid instance. However, it is not true on the abstract syntax level. First, it must be allowed that a *Subtask* can be dropped onto a *Queue*. Second, on insertion, the tool must disconnect the containment between the *Subtask* and a *Queue*. Finally, the tool must connect the inserted *Task* to the existing *Queue* and *Subtask*. In general, it is not true that a model with transitive containment can be extended to a valid instance by adding elements only.

We can state that a model allowing transitive containment with respect to its metamodel is not necessarily a partial instance. An example preceded the statement that proves that there exists a case where the transitive containment violates the rules of partial instantiation. If a model uses transitive containment where the metamodel allows it, then the model is obviously still a partial instantiation. Also, the multiplicities along the containment path are not considered in Definition of *Transitive containment*, which can lead to another type of violation.

Definition (Dangling edge relaxation). Dangling edges are edges that are not connected to a node at either end. Assuming two nodes with an edge between them, and this construct conforms to the metamodel. Dangling edge relaxation means that either of the nodes is allowed to be omitted.

Definition (Incomplete attributes). Incomplete attributes are uninitialized attributes in a model element with no default value.

In Figure 5-2, two invalid partial instances of our *DomainServers* metamodel (Figure 3-2) are presented. In Figure 5-2a, the *Queue* is associated with two *Server* nodes. In Figure 5-2b, there is only one *DB Task*. Without the deletion and modification of certain elements, these model patterns cannot be extended to a complete instance model.

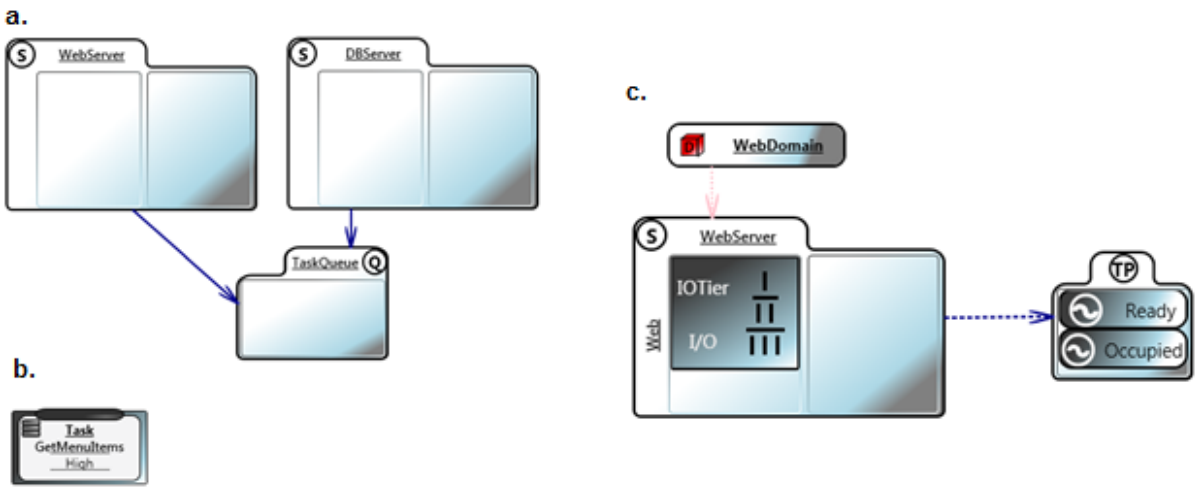


Figure 5-2 Example of (a-b) invalid partial instances, (c) valid partial instance

Figure 5-2c depicts a valid partial instance of the *DomainServers* metamodel. A server node contains a tier. The server has a *ThreadPool* containing two threads, but the required *Queue* is missing. This partial instance can be extended to a complete instance model by only adding new elements and providing uninitialized attribute values.

The following statement shows further connections between the partial instantiation and the pattern constructs presented above. If a model is created violating the instantiation rules in the following respects only: dangling edge relaxation, and incomplete attributes, then the model is a partial instantiation of its metamodel.

Relaxed instantiation relationship and relaxed multiplicity rules based on the above definitions support the definition and utilization of domain-specific design patterns and model patterns.

The practical relevance of domain-specific model patterns, and the theoretical foundations for the issue, i.e. that patterns are not always regular instances of the metamodel motivated us to provide tool support for model patterns.

Defining domain-specific model patterns and reusing them in different domain-specific models offer great perspectives for rapid application development and keep reliability at a high-level as well. VMTS has been prepared to provide tool support to create general but customizable model patterns. The support of domain-specific patterns also highlights the benefit of metamodeling.

An important rule is that patterns can be applied for models having the same metamodel as the patterns. It is not enough to create patterns on their own; there is a natural need for the capability of organizing them into pattern repositories and attaching some meta-information to the patterns as well. In this way, a large number of patterns can be handled effectively. For this purpose, we have built a DSL with metamodeling techniques. We have created pattern models that are independent from their target model into which the patterns are inserted. For the catalog, a pattern metamodel is created. As a result, a pattern repository domain-specific modeling language is created. Its nodes contain references to pattern models. Thus, the models of this language behave as pattern repositories as they can contain numerous references to real design pattern models.

Note that pattern models use the same metamodel as the target model. When inserting a pattern into a model, we offer only those patterns that have the same meta-model as the model itself.

5.3 Validating Domain-Specific Properties of Software Models

Domain-specific rules and properties provide the essence of the domain itself. Methods and model processors effectively supporting domains are based on this domain-related context.

Appropriate validation requires determining the accuracy of a model transformation, meaning the output models of the transformation satisfy certain domain-related conditions. This section discusses an approach that is based on the output model-related domain-specific properties. We extend model transformations with additional transformation rules in order to verify/validate domain-specific properties. Figure 5-3 summarizes the method and the related algorithms we worked out to validate the domain-specific properties of software models.

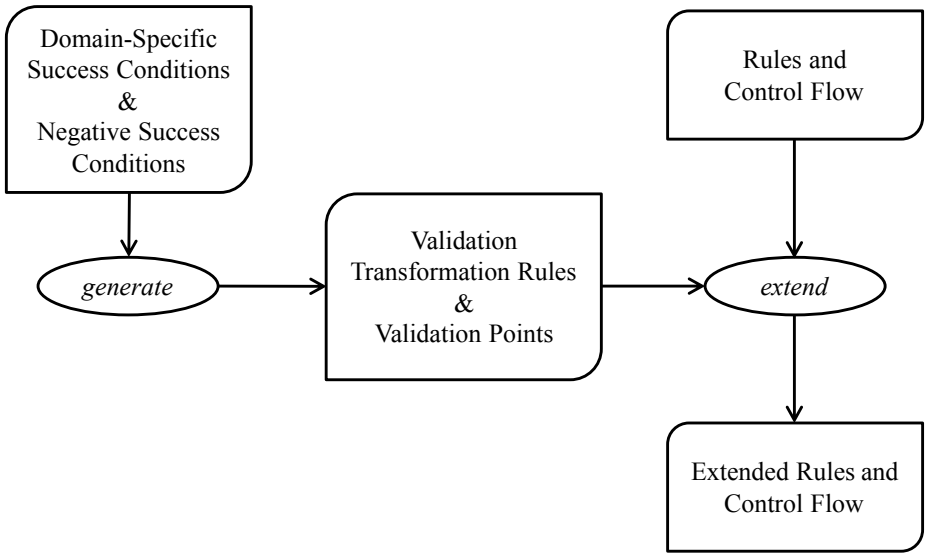


Figure 5-3 Validating domain-specific properties of software models

The goal is to require optional domain-specific properties from the output (modified or generated) software models. Therefore, we introduce the concepts of *Success Condition (SC)* and *Negative Success Condition (NSC)* [27].

Definition (Success Condition (SC)). A Success Condition defines a requirement that should be true for the output model of a model transformation rule.

Definition (Negative Success Condition (NSC)). A Negative Success Condition defines a requirement that must not be true for the output model of a model transformation rule.

There are different types of Success Conditions and Negative Success Conditions (SCs/NSCs):

1. *Constraint-based SC/NSC.* This type of SCs/NSCs is defined either in the metamodel of the output model or in the model transformation. These constraints also include attribute value and multiplicity constraints.
2. *Pattern-based SC/NSC.* This NSC is similar to the concept of Negative Application Condition (NAC) in rewriting rules [Habel et al, 1996]. Both NAC and NSC define a pattern. NAC specifies what pattern should not be found before the execution of the transformation rule. Contrarily, NSC defines the pattern that should not be found after the execution of the transformation rule. If the pattern defined by an NSC is present in the output model, then the transformation fails. Similarly, if a pattern defined by an SC is not present in the output model, then the transformation fails.
3. *Hybrid SC/NSC.* Requirements expressed by hybrid SCs/NSCs combine constraint-based SCs/NSCs with pattern-based SCs/NSCs.
4. *General SC/NSC.* This type of SCs/NSCs requires that a transformation validates or guarantees not only local, but also global output-model related requirements as well. General SCs/NSCs could be constraint-based SCs/NSCs, pattern-based SCs/NSCs, or hybrid SCs/NSCs.

Corollary. SC and NSC definitions require that the conditions should be/must not be satisfied for the output model of a transformation rule. If the selected rule is the last rule in the control flow model of the transformation, then the conditions are stated against the whole transformation.

5.3.1 Examples for Validation-related Requirements

In order to support a better understanding of the SCs and NSCs, based on our sample domain-specific language (*DomainServers*, Section 3.2.1), few examples are provided.

Define the number of specific type elements in the model (General SC). E.g., define the maximum number of *Server* type nodes in the model, or define the minimum number of *Queue* type nodes, based on the number of *Server* type nodes. These are cardinality issues related to the whole model.

```
context Server inv serverCardinality:
Server.allInstances()->count() < 40

context Queue inv queueCardinality:
Queue.allInstances()->count() >= Server.allInstances()->count()
```

At least one of the servers should be ready (Hybrid SC). This SC is similar to the following NSC: *This is not allowed that all of the servers be occupied (Hybrid NSC).*

```
context Server inv serverLoad:
Server.allInstances()->exists(s | s.Load < 80 and
    s.ThereadPool.Threads->count(t | t.State = State::Ready) >= 4)
```

The constraint requires at least one *Server* type node in which the *Load* is under 80% and has at least four *Threads* in *Ready* state.

In case of a large ThreadPool, separated CPU and I/O tiers are required (Hybrid SC). This SC defines that for each server, if the number of threads in the *ThreadPool* exceeds 50, then separated CPU and I/O tiers are employed.

```
context Server inv largeThreadPools:
Server.allInstances->forall(s | s.ThreadPool.Threads->count()<= 50 OR
     (s.Tiers->exists(t | t.Type = Type::CPU) AND
      s.Tiers->exists(t | t.Type = Type::I/O)))
```

Attribute value constraint (Constraint-based NSC). An example of attribute value constraint is the following *queueLimit* constraint. It is an invariant type constraint for the *QueueLimit* attribute of *Queue* type nodes.

```
context Queue inv queueLimit:
QueueLimit < 1200
```

These invariant type constraints can be defined as SCs/NSCs or as metamodel constraints. Validation points assign SCs/NSCs to selected transformation rules. Therefore, these assignments control where to validate these conditions. However, in the case of metamodel constraints, the transformation system must require the constraints to always be satisfied during the transformation. Furthermore, the situation is more complex if the source and the target metamodels are different. In this case, during the transformation, the processed model could contain either elements from both of the metamodels, or helper elements with general type (a metatype which is neither present in the input nor output metamodel). The goal of this is to temporarily associate nodes from the source and the target models in order to support the transformation process. The next section discusses possible approaches to ease the rigid restrictions stated by metamodel constraints.

5.3.2 Extending the Transformation with *Success* and *Negative Success Conditions* to Validate Domain-Specific Properties

The method *extending the transformation with additional transformation rules (SCs and NSCs) to validate domain-specific properties* automatically creates transformation rules that implement SCs and NSCs, and performs the validation. The original model transformation definition is extended with

generated transformation rules. As a result, the extended transformation performs the required validation. This method supports the validation of domain-specific properties (Figure 5-3).

With SCs and NSCs, we can define output model related domain-specific requirements. Now we discuss how to generate the necessary validation rules for different types of SCs/NSCs. First, we provide the definitions of the *Validation transformation rule* and the *Validation point*, and then introduce the different scenarios of the transformation extension. Next, we provide a method to generate validation transformation rules from SCs/NSCs (Section 5.3.2.1). Finally, we discuss the topic of extending model transformations with generated validation transformation rules (Section 5.3.2.2). We introduce algorithms for generating validation transformation rules, as well as extending model transformations with these rules.

Definition (Validation transformation rule). A validation transformation rule validates one or more output, model-related, domain-specific properties.

The exact points in a model transformation, where certain SCs and NSCs should be validated, are defined by validation points.

Definition (Validation point). A validation point designates a transformation rule in a model transformation control flow and refers an SC or an NSC that should be validated following the selected model transformation rule.

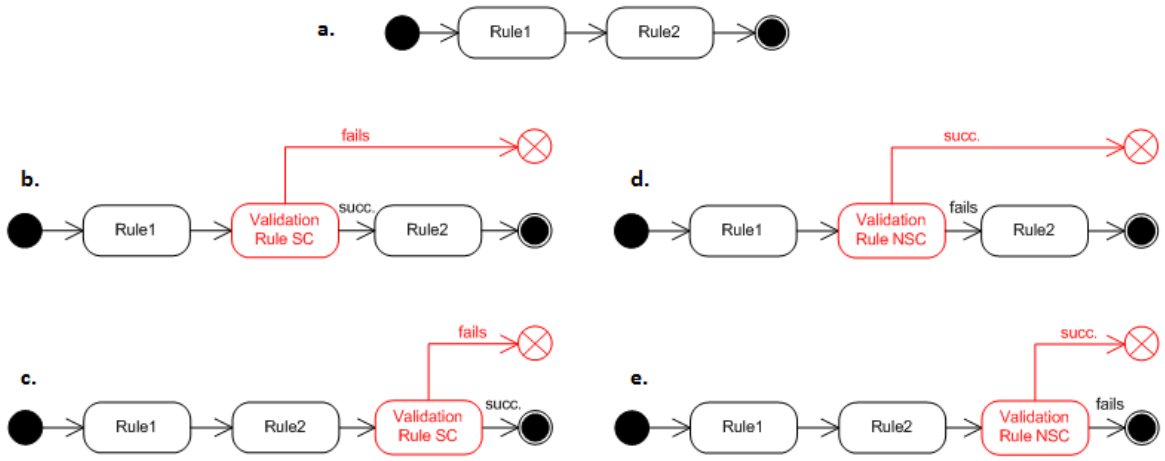


Figure 5-4 Extending model transformations with validation transformation rules: (a) Original model transformation, (b) Transformation extended with an intermediate SC, (c) Transformation extended with a final SC, (d) Transformation extended with an intermediate NSC, (e) Transformation extended with a final NSC.

Figure 5-4 introduces the scenarios in which model transformations can be extended with validation transformation rules implementing SCs and NSCs. In Figure 5-4a, the original model transformation, which has two transformation rules executed in a sequence, is presented. In Figure 5-4b, the transformation is extended with a new validation transformation rule. The new rule is inserted after *Rule1* and in the case of successful validation, according to the original control flow, it is followed by *Rule2*. In the case of failed validation, the transformation process stops with error. The scenario depicted in Figure 5-4c is similar to the previous one, but in this case, the validation transformation rule is inserted as the last (final) rule of the transformation. Scenarios shown in Figure 5-4d and Figure 5-4e are similar to the scenarios *b* and *c*, but differ in that the NSCs are implemented by validation transformation rules. Therefore, if a match can be found and the conditions of the NSC hold, the transformation should terminate with error, otherwise the control flow continues.

Definition (Validate Success Condition). A transformation *T* validates a success condition *SC* for an input model *H* if executing the transformation *T* for the input model *H* results in an output model *M* that satisfies the conditions defined by the success condition *SC*.

Definition (Validate Negative Success Condition). A transformation *T* validates a negative success condition *NSC* for the input model *H* if executing the transformation *T* for the input model *H* results in an output model *M* that does not satisfies the conditions defined by the negative success condition *NSC*.

Definition (Validated model transformation). A model transformation *T* is validated if output models generated by transformation *T* satisfy a set of success conditions and/or negative success conditions.

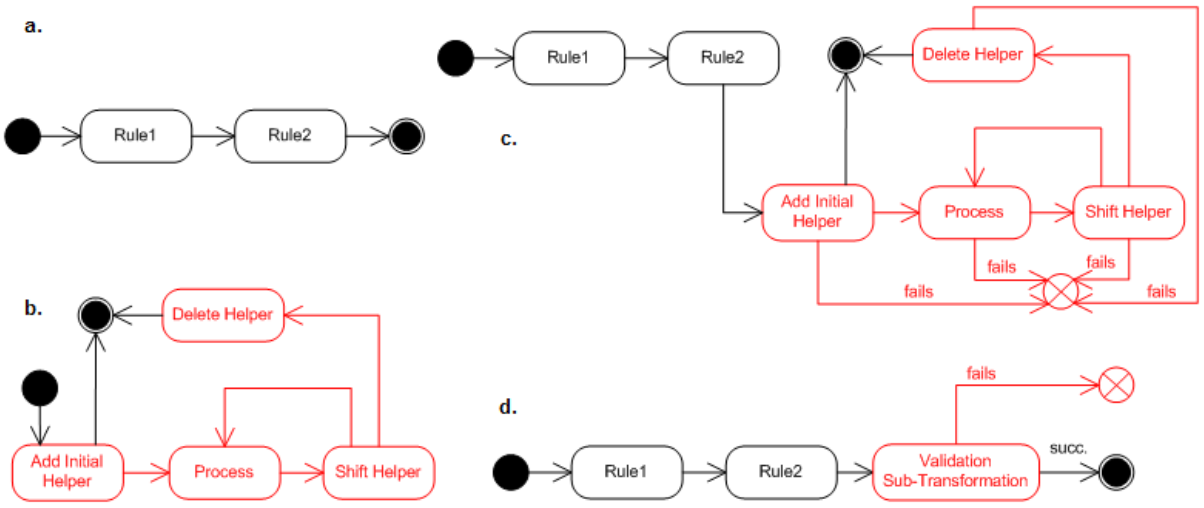


Figure 5-5 Extending model transformations with complex validation: (a) Original transformation, (b) Transformation implementing an optimized transitive closure, (c) Transformation extended, with several transformation rules, (d) Transformation extended with a sub-transformation.

In order for a model transformation be able to validate the conditions defined by SCs and NSCs, the SCs and NSCs should be implemented by transformation rules and should be inserted into the transformation control flow. Different SC and NSC types require different treatment to prepare for transformation validation.

1. *Constraint-based SC/NSC.* A method (algorithm) is required that generates validation transformation rules from constraint-based SCs/NSCs.
2. *Pattern-based SC/NSC.* The pattern of this type of SCs/NSCs can be used as both the LHS and the RHS of the appropriate validation transformation rule.
3. *Hybrid SC/NSC.* The pattern of this type of SCs/NSCs can be used as both the LHS and the RHS of the appropriate validation transformation rule. The constraints should be propagated to the LHS of the validation transformation rule.
4. *General SC/NSC.* We know that a general SC/NSC can be constraint-based, pattern-based, or hybrid. General requirements that can be expressed with a single pattern or constraint can be treated according to the previous points. In the case of more complex validation issues, e.g. one that requires several rules to be implemented, different methods should be applied. Some examples include traversing an inheritance hierarchy, within a UML class model, or identifying whether an output model contains a directed loop.

   These situations should be handled with several transformation rules, in which aggregated behavior provides the required validation. In this case, we extend the original transformation

with several transformation rules or with a sub-transformation (if the control flow language provides the feature of the hierarchical control flows). The structure of this scenario is introduced in Figure 5-5. The original model transformation contains two transformation rules. Figure 5-5b depicts a transformation in which the execution of a loop implements an optimized transitive closure traversing algorithm. In Figure 5-5c, our original transformation is extended with the validation implementing the transitive closure in inline mode. Figure 5-5d provides the same functionality while calling a sub-transformation. In this case, the validation is implemented in the form of a separated model transformation.

5.3.2.1 Generating Validation Transformation Rules

We use the algorithm GENERATEVALIDATIONTRANSFORMATIONRULE to generate validation transformation rules from constraint-based SCs/NSCs. The input parameter (*condition*) of the algorithm is an SC or an NSC. The algorithm creates a pattern with a single node, in which the metatype corresponds to the context type of the constraint (line 2-3). The algorithm traverses the navigation paths of the constraint. For each navigation step, the loop creates a new node and links it to the actual pattern (line 4-7). Next, based on the pattern, a transformation rule is created: both the LHS and the RHS of the rule is initialized with the pattern (line 8). The original constraint is propagated to the LHS of the rule (line 9). The constraint is propagated to the node that is created based on the context type of the constraint. Finally, the generated validation transformation rule is returned.

Algorithm. Pseudo code of the GENERATEVALIDATIONTRANSFORMATIONRULE algorithm

```
01: GENERATEVALIDATIONTRANSFORMATIONRULE (Constraint condition) : TransformationRule
02: RuleNode ruleNode = CREATERULENODE(condition.ContextType)
03: Pattern pattern = InitializePattern(ruleNode)
04: for all NavigationStep navigationStep in condition do
05:   Node node = CREATENODE(navigationStep.DestinationNode.MetaType)
06:   LINKNODE(pattern, node)
07: end for
08: TransformationRule rule = CreateTransformationRule(LHS = pattern, RHS = pattern)
09: PROPAGATECONSTRAINT(rule.LHS, ruleNode, condition)
10: return rule
```

Figure 5-6 introduces three constraints and their corresponding validation rewriting rules created by the algorithm GENERATEVALIDATIONTRANSFORMATIONRULE. The context type of the constraints is *Server* for all of the example constraints. In the case of constraint *serverLoad*, the path *Server - ThreadPool - Thread* should be traversed. Similarly, the paths covered by the constraint *largeThreadPools* are *Server - Tier*, *Server - Tier*, and *Server - ThreadPool - Thread*. In the case of constraint *serverQueueThreadNumbers* the traversed paths are *Server - ThreadPool - Thread* and *Server - Queue*.

Validation transformation rules validate output, model-related, domain-specific properties thus, they instantiate the output metamodel. Note that the LHS and the RHS graphs of the generated validation transformation rule have the same pattern. Given this structure a validation rule does not insert or delete any node or edge from the output model. The only difference between the LHS and RHS of the generated rule is the constraint that is propagated only to the LHS.

Based on the pseudo code of the algorithm GENERATEVALIDATIONTRANSFORMATIONRULE, we can state that the validation transformation rule generated with algorithm GENERATEVALIDATIONTRANSFORMATIONRULE does not modify the output model. Furthermore, we can conclude that validation transformation rules generated with algorithm

GENERATEVALIDATIONTRANSFORMATIONRULE are not necessarily valid partial instances of the output metamodel.

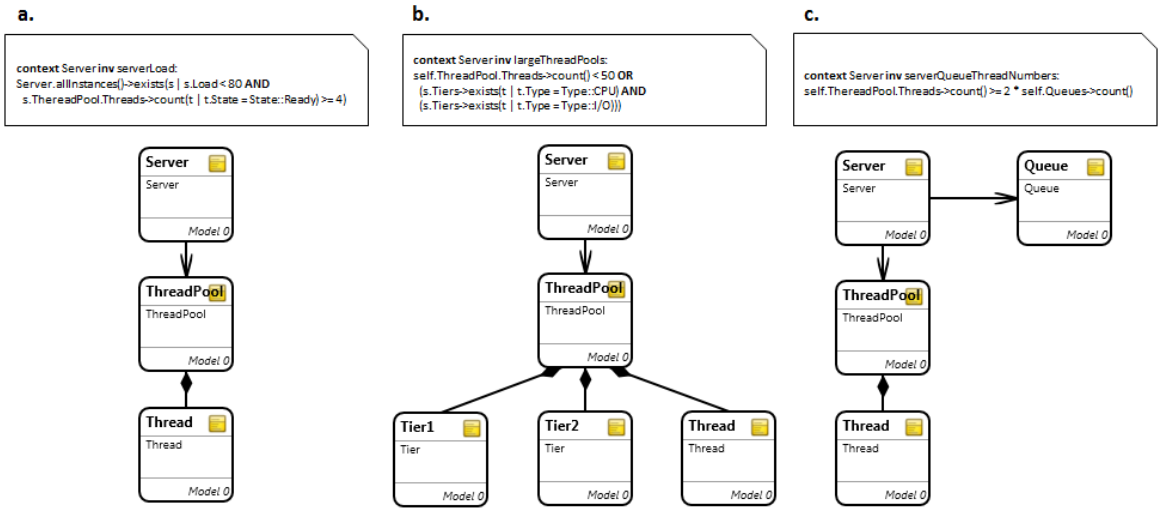


Figure 5-6 A Algorithm GENERATEVALIDATIONTRANSFORMATIONRULE: (a) Constraint *serverLoad* and the generated rule, (b) Constraint *largeThreadPools* and the generated rule, (c) Constraint *serverQueueThreadNumbers* and the generated rule.

5.3.2.2 Extending Model Transformations with Validation Transformation Rules

We use the algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES to extend the control flow model of the transformation with the generated validation transformation rules. The input parameters of the algorithm define the transformation *T* that should be extended, a condition list containing SCs and NSCs, and the list of the validation points where SCs and NSCs should be validated. The algorithm creates a flow final node (line 2) then, using a loop, processes the validation points (line 3-25). For each validation point, the SC or NSC condition is selected from the *conditionList* (line 4). Next, based on the representation type of the condition (*ConstraintBased*, *PatternBased*, or *Hybrid*), the algorithm creates a validation transformation rule. In the case of *ConstraintBased* representation type, using the constraint of the condition, the validation rule is created with algorithm GENERATEVALIDATIONTRANSFORMATIONRULE (line 6-7). Otherwise, for conditions with representation type *PatternBased* and *Hybrid* the validation rule is created based on the pattern of the condition (line 10). Both the LHS and the RHS of the new validation rule is initialized with the pattern of the condition. For *Hybrid* type conditions, the constraint of the SC or NSC is propagated to the LHS of the validation rule (line 12-14). Then the algorithm assigns the new rule to the transformation *T* (line 15), and arranges the necessary flow edge modifications (line 16-24). During the creation and rearrangement of the flow edges, we differentiate the success type of the conditions, i.e. SCs and NSCs are handled in a different way. Figure 5-7 illustrates this part of the algorithm. In both of the cases the new validation rule is inserted after the transformation rule selected by the *validationPoint*. The difference is in the type of the flow edges beginning from the validation rule. In the case of SCs, a *Fail* type edge goes to the *flowFinal* and *Success* type edges points to the original target rules or to the end rule. Contrarily, in the case of NSCs, a *Success* type edge goes to the *flowFinal* and *Fail* type edges point to the original target rules or to the end rule. Finally, the extended transformation definition is returned.

Algorithm. Pseudo code of the EXTENDTRANSFORMATIONWITHVALIDATIONRULES algorithm

01: EXTENDTRANSFORMATIONWITHVALIDATIONRULES (Transformation *T*, List *conditionList*, List *validationPointList*) : Transformation
02: FlowFinal *flowFinal = T.CreateFlowFinal()*

03: **for all** ValidationPoint *validationPoint* in *validationPointList* **do**
04: Condition *condition* = *conditionList.Get(validationPoint.ConditionReference)*
05: TransformationRule *validationRule* = NULL
06: **if** (*condition.RepresentationType == RepresentationType :: ConstraintBased*) **then**
07: *validationRule* = GENERATEVALIDATIONTRANSFORMATIONRULE(*condition.Constraint*)
08: **else**
09: // executed for PatternBased and Hybrid type conditions
10: *validationRule* = CREATETRANSFORMATIONRULE(*LHS=condition.Pattern, RHS=condition.Pattern*)
11: **end if**
12: **if** (*condition:RepresentationType == RepresentationType :: Hybrid*) **then**
13: PROPAGATECONSTRAINT(*validationRule.LHS, condition.Constraint*)
14: **end if**
15: *T.AddTransformationRule(validationRule)*
16: **if** (*validationPoint:SuccessT ype == SuccessType :: SC*) **then**
17: CREATEFLOW(*validationRule, flowFinal, FlowType :: Fail*)
18: MODIFYFLOWS(*validationPoint.TransformationPoint, validationRule, FlowType :: Success*)
19: **else**
20: // executed for NSCs
21: CREATEFLOW(*validationRule, flowFinal, FlowType :: Success*)
22: MODIFYFLOWS(*validationPoint.TransformationPoint, validationRule, FlowType :: Fail*)
23: **end if**
24: CREATEFLOW(*validationPoint.TransformationPoint, validationRule, FlowType :: Success*)
25: **end for**
26: **return** *T*

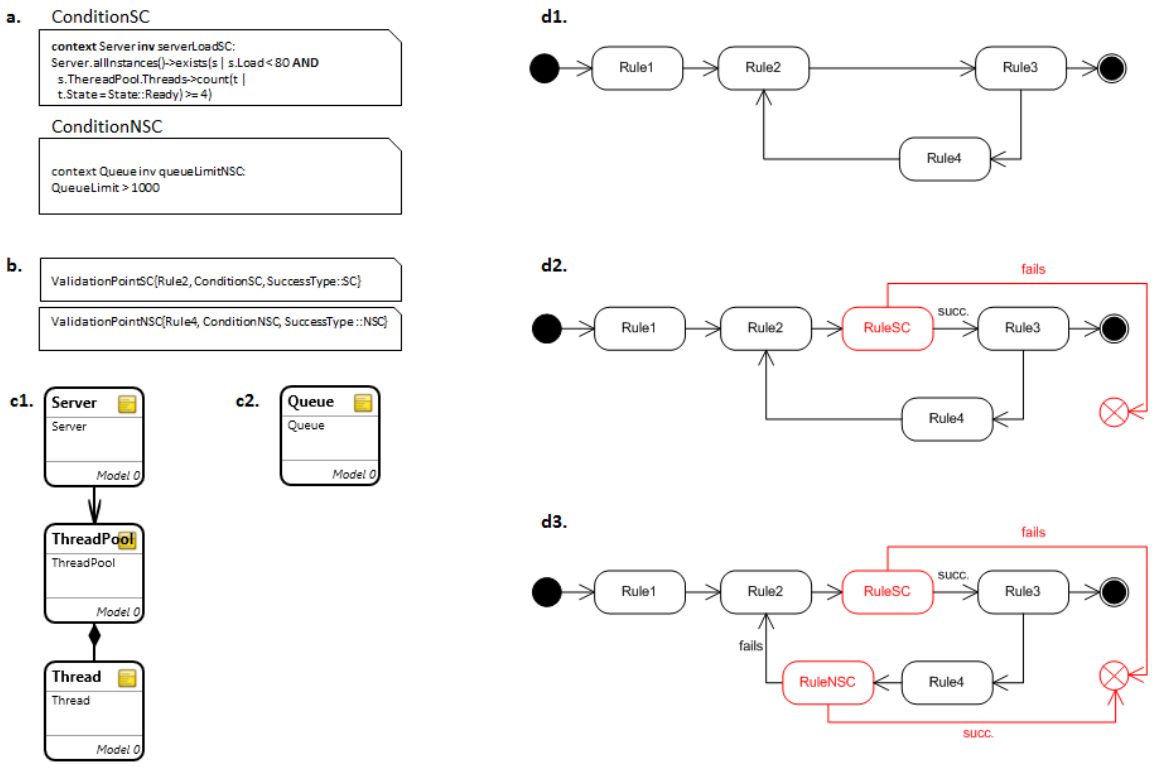


Figure 5-7 Algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES: (a) A success and a negative success condition of the transformation, (b) Validation points, (c) Generated validation transformation rules (*RuleSC* and *RuleNSC*), (d) The stages of the transformation control flow extension.

Figure 5-7 introduces the whole process of the transformation validation. Figure 5-7a defines a success and a negative success condition that are stated against the transformation. In Figure 5-7b two validation

points are introduced. These validation points define where and which conditions should be validated. Figure 5-7c1 depicts the validation rule created based on condition *ConditionSC* and Figure 5-7c2 shows the validation rule created based on condition *ConditionNSC*. In Figure 5-7d1 the original transformation is presented. Figure 5-7d2 shows the state after extending the transformation with a flow final and the validation rule *RuleSC*. The flow edge between *RuleSC* and *Rule3* is followed in the case of successful validation, otherwise the control follows the new edge starting from *RuleSC* and pointing to flow final node. Finally, Figure 5-7d3 presents the state of the control flow after extending it with the validation rule *RuleNSC*. The flow edges are connected in a different way: in the case of satisfying the conditions of *NSC* the *successful* edge is followed that points to the flow final, otherwise the control is passed to *Rule2*.

The following statements and consequences summarize statements related to (i) the presented algorithms, (ii) the transformations processed with these algorithms, and (iii) the models resulted by the extended transformations.

- Assume an input model *H*, a model transformation *T*, and a success condition *SC*. A validation transformation rule *VR* is generated from *SC* with the algorithm GENERATEVALIDATIONTRANSFORMATIONRULE. Model transformation *T* is extended with the algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES, using the validation point *VP{T.LastTransformationRule, SC, SuccessType::SC}*. The resulted transformation is *T'*. If the transformation *T* validates the success condition *SC* for the input model *H*, then transformation *T'* also validates the success condition *SC* for the input model *H*. Transformations *T* and *T'* generate the output models *M* and *M'* respectively.
- Assume an input model *H*, a model transformation *T*, and a negative success condition *NSC*. A validation transformation rule *VR* is generated from *NSC* with algorithm GENERATEVALIDATIONTRANSFORMATIONRULE. Model transformation *T* is extended with algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES using the validation point *VP{T.LastTransformationRule, NSC, SuccessType::NSC}*. The resulted transformation is *T'*. If the transformation *T* validates the negative success condition *NSC* for the input model *H*, then transformation *T'* also validates the negative success condition *NSC* for the input model *H*. Transformations *T* and *T'* generate the output models *M* and *M'* respectively.
- Assume an input model *H*, a model transformation *T*, a success condition *SC* and a validation point *VP{T.LastTransformationRule, SC, SuccessType::SC}*. Based on the success condition *SC*, a validation rule *VR* is generated with algorithm GENERATEVALIDATIONTRANSFORMATIONRULE. The transformation *T* is extended by algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES, applying the validation point *VP*. The resulted transformation is *T'*. If the transformation *T'* finishes successfully for input model *H*, generating the output model *M'*, then the original transformation *T* validates the success condition *SC* for input model *H* while generating the output model *M*.
- Assume an input model *H*, a model transformation *T*, a negative success condition *NSC*, and a validation point *VP{T.LastTransformationRule, NSC, SuccessType::NSC}*. Based on the negative success condition *NSC*, a validation rule *VR* is generated with algorithm GENERATEVALIDATIONTRANSFORMATIONRULE. The transformation *T* is extended by algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES applying the validation point *VP*. The resulted transformation is *T'*. If transformation *T'* finishes successfully for input model *H* generating the output model *M'*, then the original transformation *T* validates the negative success condition *NSC* for input model *H* while generating the output model *M*.

- Assume that a transformation *T* terminates for an input model *H*. Applying the algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES we extend the transformation *T* with one or more validation transformation rules. The resulted transformation is *T'*. Transformation *T'* terminates for input model *H*.
  We can see that these types of rules (do not perform modification) and this type of transformation extension (no additional loops and the inserted rules are performed once in a sequence) do not make the originally terminating transformation *T* non-terminating.
- Assume an input model *H*, a model transformation *T*, a success condition *SC*, a negative success condition *NSC*, and two validation points VP_{SC}{*T.LastTransformationRule, SC, SuccessType::SC*} and VP_{NSC}{*T.LastTransformationRule, NSC, SuccessType::NSC*}. The generated validation rules for *SC* and *NSC* are R_{SC} and R_{NSC} respectively. The transformation *T* is extended by the algorithm EXTENDTRANSFORMATIONWITHVALIDATIONRULES applying the validation points VP_{SC} and VP_{NSC}. The resulted transformation is *T'*. Transformations *T* and *T'* are executed for the same input model *H*, the output models are *M* and *M'* respectively. If the rules R_{SC} and R_{NSC} successfully validate the conditions, i.e. *SC* is present in the model and *NSC* cannot be found in the model, then the resulted output models *M* and *M'* are identical.
  Note that in the case of validation error the output models *M* and *M'* can be different. The reason is that unsuccessful validation rules stop the transformation execution and all rules that follow an unsuccessful validation rule in the control flow model are not executed.

This section has discussed a way of validating graph rewriting-based model transformations. We have investigated the validation-related possibilities and have developed a novel approach for the verification and validation of output, model-related, domain-specific properties. We have provided algorithms for generating validation transformation rules and extend model transformations with the generated validation rules. Finally, we have provided several statements regarding the discussed algorithms, the transformations processed with these algorithms and the models resulting from extended transformations.

5.4 Modularized Constraint Management

The precise definition of graph rewriting-based model transformation requires that beyond the topology of the rules, further textual constraints to be added. These constraints often appear repetitively in a transformation; therefore, constraint concerns crosscut the transformation. It is conductive to define often applied constraints as physically separated modules and indicate the places where to use them. This effort provides solutions to structuring, modularizing and propagating repetitively occurring and crosscutting constraints. We propose an aspect-oriented approach that allows for consistent constraint management; in which repetitive and crosscutting constraints can be semi-automatically identified.

Often, we require validating several rules or whole transformations, which may cause the same constraint concerns to appear numerous times in a transformation. Regarding this recurrence of constraint concerns, it is beneficial to distinguish between the classical constraint repetition and the crosscutting constraints. According to [Sutton and Rouvellou, 2002], the definition for the term *concern* is, "any matter of interest in a software system".

The classical constraint repetition is similar to the frequently appearing lines of program code in a source file (also known as, code clones). In the source code domain, this problem is handled with program segmentation. In most cases, it is implemented with functions; the recurring lines of source code are placed into a function and the function is then called from the appropriate position. This method can be

applied to model transformation constraints as well. This can be achieved by extracting the repetitive constraints into separated components and, similarly to function calls, we designate those points in the model transformation where they will be applied.

Regarding crosscutting concerns, the situation is significantly different. As opposed to repetitions, crosscutting concerns of a design cannot be modularly separated. If a concern attempts to decompose, according to a specified design principle, other concerns will crosscut this decomposition. This implies that crosscutting is relative to each particular decomposition.

To summarize crosscutting concerns, there is no way to achieve a modular design. In the case of repetitive constraints, consistent constraint management is difficult. In order to mitigate these issues, our aim is to physically separate the different concerns, namely, the structure of the transformation rules and constraints, and design them separately. Next, using a weaving mechanism, we generate the executable artifact that combines the two concerns. This generated representation, containing both repetitive and crosscutting constraints is similar to a binary file compiled from source code and is not edited by the transformation engineer. Therefore, no problems arise, despite the generated artifact concerns not being separated.

The approach presented in this section provides solutions for both repetitive and crosscutting constraints. The novel results, provided by this approach are: (i) the distinction of repetitive and crosscutting constraints in model transformations, (ii) the mechanism that handles the repetitive constraints and (iii) a generalized, semi-automatic identification of repetitive and crosscutting constraints.

Figure 5-8 introduces the main elements and steps of the modularized constraint management, i.e. summarizes the method and algorithms for handling the validating constraints in a modular way [Lengyel, 2013].

5.4.1 Managing Repetitive Constraints

In our approach, model transformation-related problems regarding validation constraint management are separated into two groups: namely, the management of repetitively appearing constraints and the management of crosscutting constraints. This section clarifies the differences between these two types of constraints and discusses the methods applied for the handling of repetitive constraints.

In software engineering, it is advisable to follow the separation of concerns [Dijkstra, 1976] (SoC) principle. In essence, this indicates that, in dealing with complex problems, the only possible solution is to divide the problem into sub-problems, and then solve them separately. Next, combine the partial solutions to create a complete solution. One type of concerns, such as rewriting rules, may smoothly be encapsulated within building blocks, by means of conventional techniques of modularization and decomposition, whereas the same is not possible for other types. More specifically, these types crosscut the design and are therefore called crosscutting concerns. Because of their specialty, crosscutting concerns arise two significant problems:

- The *scattering problem*: the design of certain concerns is scattered over many building blocks.
- The *tangling problem*: a building block can include the design of more than one concern.

Recall that in the validation of model transformations, there are two concerns: the functionality of the transformation and the constraints ensuring the validation. Sometimes modularizing one of the two concerns implies that the other concern will crosscut the transformation, and vice versa.

Both scattering and tangling have several negative consequences for the transformations they affect. However, the aim of aspect-oriented methods is to alleviate these problems by modularizing crosscutting concerns. Therefore, in the case of crosscutting constraints, aspect-oriented methods should be applied

in order to achieve consistent constraint management. Both logically coherent constraints (crosscutting constraints) and repetitively appearing constraints should be physically maintained in a modularized manner.

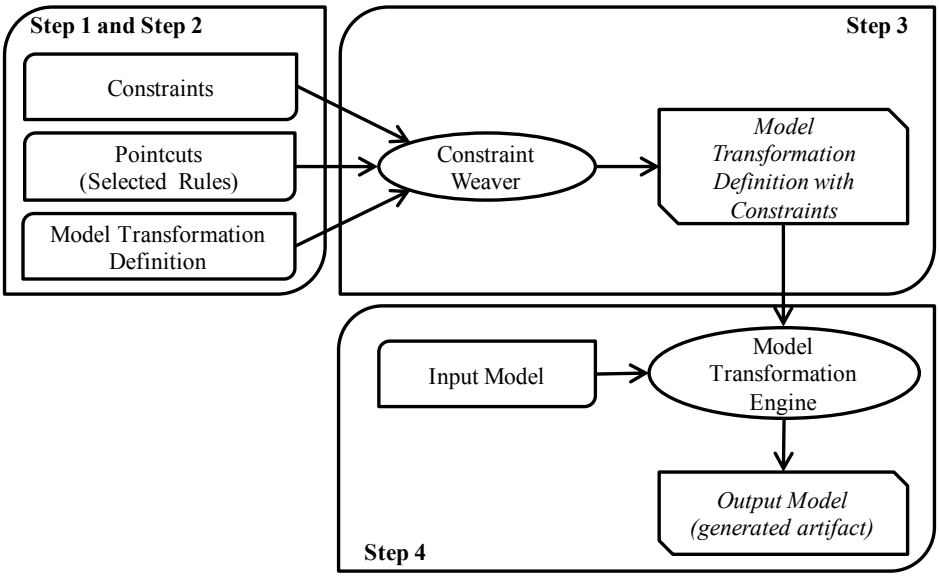
For the problem of crosscutting constraint management, a solution has been provided in [Lengyel, 2006] and this solution has been summarized in Section 3 of the current thesis. Current section provides a novel approach for handling repetitive constraints in model transformations.

5.4.1.1 The Constraint Management Process

As we previously stated, consistent constraint management requires a mechanism that supports the handling of repetitive constraints. Our approach provides the following methods regarding their management:

- Constraints are defined independently from transformation rules. This allows us to maintain the constraints in a physically separated place.
- Constraint calls are defined along with the designation where the constraints should be applied. Using the generalized version of the Global Constraint Weaver, the approach automatically assigns the constraints to the indicated points of the transformation.

In this approach, the selection of the rules, where the aspect should be propagated (constraint calls), is performed manually by the transformation designer. This method is supported by the weaver tool: the potential transformation rule nodes are offered for the transformation designer, who can manually select those which are required.



Step 1. Defining and maintaining constraints and transformations
Step 2. Selecting the appropriate rewriting rules (supported by type-based searching)
Step 3. Weaving the constraints to the transformation rules
Step 4. Executing the transformation

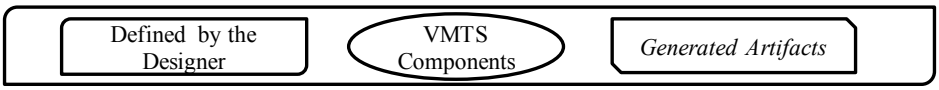


Figure 5-8 Managing validating constraints in a modular way

The whole process of repetitive constraint handling, and its role in the model transformation, is illustrated in Figure 5-8. Related to this process, we have identified four steps:

1. *Defining and maintaining constraints and transformations.* This step is performed by the transformation designer.
2. *Selecting the appropriate rewriting rules.* This step is also completed by the transformation designer. The result of this step is the constraint calls that designate the rewriting rules where to propagate the constraints (from where the constraint validation should be called during the transformation).
3. *Propagating the constraints to the rules.* This step is executed by the weaver component. The weaving method gets the transformation, the constraints, and the constraint calls. The result of the weaving process is the transformation definition with the assigned constraints.
4. *Executing the transformation.* This step is performed by the model transformation engine. The inputs are the transformation definition that contains the constraints and the input model. The output of the model transformation is the generated artifact that can also be a model or optional text, e.g. source code.

5.4.1.2 Generalizing the Constraint Weaving

We have worked out the generalized constraint weaving mechanism. This Generalized GCW (GCW2) method supports the weaving of the following constraint constructs:

- Aspect-oriented constraints driven by weaving constraints,
- Repetitive constraints driven by their constraint calls.

Aspect-oriented constraints are OCL constraints, we separate them physically from transformation rules. Weaver algorithms weave them into the rules. The context information of the aspect-oriented constraints is used as a type-based pointcut. This pointcut, based on the metatype information, selects the appropriate rule nodes. This weaving process is referred to as *type-based weaving*. In order to further develop the weaving procedure, we apply weaving constraints. A weaving constraint is similar to a property-based pointcut. This is also an OCL constraint, which restricts the type-based weaving. Obviously, weaving constraint is not added to. Weaving constraints allow for the verification of optional conditions during the weaving process. We refer to it as *constraint-based weaving*. The physically separated constraints require a weaver that applies type-based and constraint-based weaving mechanisms, and facilitates the assignment of constraints to transformation rules.

In generalized constraint weaving approach, aspect-oriented constraints and repetitive constraints both represent constraints, which are defined separately from model transformations. They are handled separately, because their weaving is driven by different constructs. The weaving of aspect-oriented constraints is supported by the weaving constraints and the weaving of repetitive constraints is driven by constraint calls. Therefore, these two types of constraints are not mixed.

The inputs of the GCW2 algorithm include the transformation definition, the aspect-oriented constraints with their weaving constraints, and the repetitive constraints with their constraint calls. The output of the weaver is the constrained transformation. Algorithm GLOBALCONSTRAINTWEAVER2 depicts the pseudo code of the GCW2 algorithm.

Algorithm. Pseudo code of the GLOBALCONSTRAINTWEAVER2 algorithm

```
1:  GLOBALCONSTRAINTWEAVER2 (Transformation T, ConstraintList AOCs, ConstraintList, weavingCs,
      ConstraintList repetitiveCs, ConstraintCallList constraintCalls)
2:  for all Constraint AOC in AOCs do
3:    for all TransformationRule R in T do
4:      nodesWithProperMetaT ype = GETNODESBYMETATYPE (context type of AOC, R)
5:      nodesWithProperStructure = CHECKSTRUCTURE (nodesWithProperMetaT ype, R, AOC)
```

```
6:        checkedNodes = CHECKWEAVINGCONSTRAINTS (nodesWithProperStructure, weavingCs)
7:        WEAVECONSTRAINT (AOC, checkedNodes)
8:     end for
9:   end for
10: for all Constraint RC in repetitiveCs do
11:    for all ConstraintCall CC in constraintCalls do
12:       nodesToWeave = EVALUATECONSTRAINTCALL (CC, RC)
13:       WEAVECONSTRAINT (RC, nodesToWeave)
14:    end for
15: end for
```

The GCW2 algorithm is passed through a model transformation, a list of aspect-oriented constraints, a list of weaving constraints, a list of repetitive constraints and a list of constraint calls. The algorithm, using type-based weaving and applying weaving constraints, weaves the aspect-oriented constraints to the appropriate nodes of the rules. Furthermore, the algorithm weaves the repetitive constraints to the rules designated by the constraint calls.

The GCW2 algorithm uses different blocks to manage the aspect-oriented constraint weaving (line 2-9), and the repetitive constraint weaving (line 10-15). In the first block, for each aspect-oriented constraint and transformation rule pair, the algorithm identifies the possible places where the constraint can be woven. It then checks the surrounding structures of these locations and evaluates the weaving constraint for the appropriate places. Finally, the constraint is woven to the correct rules. In the second block, for each repetitive constraint and constraint call pair, the algorithm decides where to weave the actual repetitive constraint, and then performs the weaving.

The proposed method for handling repetitive constraints facilitates the definition of constraints independent of transformation rules and designates the rewriting rules, i.e., where to apply them. The approach automatically weaves the constraints to the designated points in the transformation. The benefit of this approach is that the constraints are maintained in one place and in one copy. Furthermore, our method supports a better understanding of both the transformations and constraints.

This section introduced the GCW2 algorithm, which facilitates the constraint weaving driven by both weaving constraints and manually defined constraint calls. The next section discusses the method to modularize transformation constraints if they already exist in model transformations.

5.4.2 Semi-Automatic Modularization of Transformation Constraints

In [20], a mechanism is introduced for systematically identifying crosscutting constraints. This section provides a generalized, semi-automatic method for modularizing both repetitive and crosscutting constraints.

In model transformations, some validation or other concerns can be expressed by several constraints. These concerns (expressed by more than one constraint) are the source of the crosscutting. In our approach, transformation designers can aggregate constraints into groups, in which each group represents a concern. The examples provided are the *syntactic well-formedness* and the *semantic well-formedness* groups.

Group_SyntacticWellFormedness {*DanglingEdges1, DanglingEdges2, ClassAndItsParentAreTheSame*}

Group_SemanticWellFormedness {*MultipleInheritance, CheckInternalCondition, CheckSealedCondition*}

The inputs of the modularization method are the transformation itself and the grouping definitions. The expected outputs are the modularized constraints and the constraint calls that support the weaving process. For transformation designers, the modularized constraints are represented as physically modularized concerns with the goal to support the management of validation requirements. The tasks required by the modularization method are as follows:

1. Collect the constraints from the transformation.
2. Identify the crosscutting constraints.
3. Identify the repetitive constraints.
4. Extract the crosscutting constraints as aspects, and generate the constraint calls to support their weaving.
5. Extract the repetitive constraints as aspects, and generate the constraint calls to support their weaving.

In Step 2, the identification of crosscutting concerns is supported by the grouping definition. The algorithm checks whether the semantically coherent concerns are, physically, in the same rule or scattered across several rules. Concerns represented by single constraints cannot crosscut the transformations, but if they are appearing several times they are classified as repetitive constraints.

The crosscutting constraint identification method provides the coloring and extracting algorithms. These algorithms have been updated to support both the repetitive and crosscutting constraint modularization in a general way. Based on the groups and the identified concerns, the reworked coloring algorithm assigns different colors to the concerns of the transformation. The automatic concern identification also accounts for the constraints not appearing in any of the user defined groups. In the output of the coloring algorithm, each color represents a concern. These concerns should be modularized with the goal of effective management. After the coloring, the extracting algorithm creates aspects from crosscutting and repetitive constraints, as well as generates the constraint call definitions.

The subsequent sections elaborate upon the algorithms, and their operation is illustrated with the help of our case study.

5.4.2.1 Generalized Coloring Algorithm

The algorithm gets the transformation with its constraints and the grouping definitions. The expected result is a concern list and a coloring table, which provides the transformation rule and affected concern relations.

A concern is represented by a color and can be an optional condition or property expressed by one (simple) or several constraints. Examples of this include the well-formedness concerns of our case study as well as more simplified versions, namely, those including an attribute value or the existence of adjacent nodes of a specific type.

The next algorithm shows the pseudo code of the COLORING algorithm. The model transformation *T* and its corresponding groups are passed to the algorithm. The algorithm creates a list of rule-constraint pairs. The list contains each transformation rule-constraint pair assignment, defined in transformation *T*. Based on the rule-constraint assignments, the algorithm identifies the crosscutting concerns for each group (line 5). Next, the coloring table is updated with the actual group information, even if there exists no crosscutting related to the actual group. Then, the algorithm creates a concern (constraint) list (line 8), in which each member of the list represents a separated concern. This means that, if a constraint in the transformation contains more than one concern, the constraint is decomposed into several

constraints. Therefore, more than one list member is created from such constraints. Groups are also added to the concern list. Based on the constraints of the transformation and the list of concerns, the algorithm identifies repetitive constraints (line 12) and updates the coloring table accordingly.

Algorithm. Pseudo code of the COLORING algorithm

```
1:  COLORING (Transformation T, GroupList groups)
2:  ColoringTable coloringTable = new ColoringTable();
3:  RuleConstraintPairList ruleConstraintPairs = COLLECTRULECONSTRAINTPAIRS (T)
4:  for all Group G in groups do
5:     CrosscuttingList crosscuttings = IDENTIFYCROSSCUTTING (G, ruleConstraintPairs)
6:     coloringTable.UPDATECOLORINGTABLE (G, ruleConstraintPairs, crosscuttings)
7:  end for
8:  ConcernList concerns = COLLECTSEPARATEDCONCERNCONSTRAINTS (T)
9:  concerns.ADDGROUPS (groups)
10:  ConstraintList constraints = T.GETCONSTRAINTS()
11:  for all Constraint C in constraints do
12:    ConstraintList repetitives = IDENTIFYREPETITIVECONSTRAINTS (C, concerns)
13:    coloringTable.UPDATECOLORINGTABLE (C, constraints, repetitives)
14: end for
```

5.4.2.2 Generalized Constraint Extracting Algorithm

The algorithm gets the model transformation and the results of the coloring algorithm. The results of the algorithm are the modularized constraints and the constraint calls supporting the weaving. Modularized constraints can be represented as concerns, because transformation designers usually prefer this view to work with validation conditions.

The algorithm creates the modularized constraints based on the provided concern list. The group concerns are handled in a different way from simple constraints or constraint part concerns: each member of the group concern is modularized into a different constraint but can be edited as a concern with its member constraints. The second part of the extracting algorithm creates the constraint calls both for crosscutting and repetitive constraints. These constraint calls contain the exact list of the rules from which they should be called. In general, for modularized crosscutting constraints (aspects) we prefer to use weaving constraints instead of the constraint calls. This is because with weaving constraints more complex conditions can be defined, and this type of weaving definition is used when defining these artifacts manually. In the current case, the artifacts are created by the extracting algorithm. The goal is to provide an effective method that modularizes the concerns and creates such weaving artifacts that can reproduce the original transformation, exactly. Therefore, creating constraint calls for crosscutting constraints is the appropriate decision.

The EXTRACTING algorithm gets the transformation *T*, the concerns identified by the COLORING algorithm and the *coloringTable*. The algorithm processes the concerns in two blocks. In the first block, the group concerns (e.g. *SyntacticWellFormedness* and *SemanticWellFormedness*) are processed: each constraint, although related to the group, is independent and is added to the modularized constraint list (line 3-7). In the second block, the constraints of the non-group concerns are processed: simple constraints and constraint parts (line 8-10). Next, using the coloring table (transformation rule – concern/constraint mappings), the algorithm creates the constraint calls for each constraint.

Algorithm. Pseudo code of the EXTRACTING algorithm

```
01:  EXTRACTING (Transformation T, ConcernList concerns, ColoringTable coloringTable)
02:  ConstraintList modularizedConstraints = new ConstraintList ()
03:  for all Concern groupConcern in concerns.GroupConcerns do
04:     for all Constraint C in groupConcern do
05:        modularizedConstraints.Add (C)
06:     end for
```

```
07:   end for
08:   for all Concern nonGroupConcern in concerns.NonGroupConcerns do
09:      modularizedConstraints.Add (nonGroupConcern.Constraint)
10: end for
11: ConstraintCallList constraintCalls = new ConstraintCallList()
12: for all ColoringItem coloringItem in coloringTable do
13:    ConstraintCall constraintCall = CREATECONSTRAINTCALL(coloringItem, T)
14:    constraintCalls.Add (constraintCall)
15: end for
```

We have discussed that in graph rewriting-based model transformations, the two main concerns are functionality, defined by the transformation rules, and the validation properties, expressed through constraints. Regarding to model transformations, we have introduced the problem of repetitive and crosscutting constraints. We have identified the difference between repetitive and crosscutting constraints. We have shown that, in certain cases, crosscutting cannot be eliminated, but could be solved by applying aspect-oriented mechanisms. We have provided a mechanism for handling repetitive constraints. Unifying their treatment, we have developed a generalized method with its algorithms for semi-automatic modularization of repetitive and crosscutting constraints in model transformations.

5.5 Conclusions

This chapter has discussed both theoretical and practical methods to support domain-specific model patterns, validating domain-specific properties of software models and handling the validating constraints in a modular way.

We have introduced the term partial instantiation, and we have discussed that this construct allows to use the same metamodels for the pattern and the target model. We worked out various constructs that help relaxing the instantiation rules with the goal to support the treatment of premature model parts, i.e. patterns.

There are several modeling and metamodeling frameworks that support domain-specific user interface and transformation modeling. Based on our investigations, none of these frameworks provides tool support for domain-specific model patterns, however, the results of this area can be realized in several of them. Also, there are several design pattern description languages. Most of them aim at a formal, and more precise description than it can be achieved with UML. A comparative discussion of formal pattern languages can be found in [Flores and Fillottrani, 2003]. The primary target of these specification languages consists of object-oriented design patterns. The uniqueness of our approach is that we use the same metamodel relaxed with certain rules for the design pattern as the model in which the design pattern is going to be inserted.

Domain-specific design patterns have been successfully applied for domain-specific model processors, i.e. both for the transformation rules and the control flow models.

We have provided a method and algorithms for validating domain-specific properties of software models. We have introduced the success conditions (SCs) and the negative success conditions (NSCs), which allow to define the required domain-specific properties and rules. We have discussed the algorithms, which generate the validating constraints based on the success conditions. We have introduced further algorithms that help to automatically extend the control flow models with validating transformation rules. As a result, this approach provides a method for validating domain-specific properties of software models during the model processing.

Finally, we have worked out a method and supporting algorithms to handle *validating constraints* in a modular way. The method is able to semi-automatically identify the crosscutting constraints in model transformations.

The discussed area related selected scientific results from my research activities are summarized in Section 7.3 (Thesis III: Applying Domain-Specific Design Patterns and Validating Domain-Specific Properties).

These results allow both the effective work with domain-specific model patterns and the validation of these patterns. My results contribute to the effective, domain-specific, model-based development methods, i.e. increase the quality of the software artifacts and reduces both the development time and the amount of the necessary resources.

6 Application of the Results

Research groups are responsible to rapidly respond to industrial and technical requirements, be flexible in various engineering tasks, react with up to date courses focusing on both theoretical and practical aspects. As a university, we are exploring deeper knowledge beyond the direct needs of engineering, research and training of the engineers. As a competent research team, we are in daily contact with the industry. We listen to the industrial needs, utilize our research and development results, and provide solutions for the real requirements in the form of applications and services.

A significant part of the presented results is motivated by R&D and industrial projects. Usually industrial colleagues, university colleagues and students work together to achieve project goals. As a result, besides the realization of the project, students are trained for a valuable, industry-relevant area, furthermore, researchers collect a relevant industry-specific knowledge that can be utilized during the courses, thereby these common projects result a multiplicative effect.

Over the past decade, model processors had a significant role in software development, during the generation of software products, as well as in modification and optimization of software services. Utilizing automated model processors, we could increase the quality of software products. Both our research group and several industrial partners recognized that model transformation definitions, as they are also software artifacts, could incorporate errors. The presented novel scientific results provide verification / validation methods, furthermore, utilization and application practices that help the exploration of the remained hidden conceptual defects.

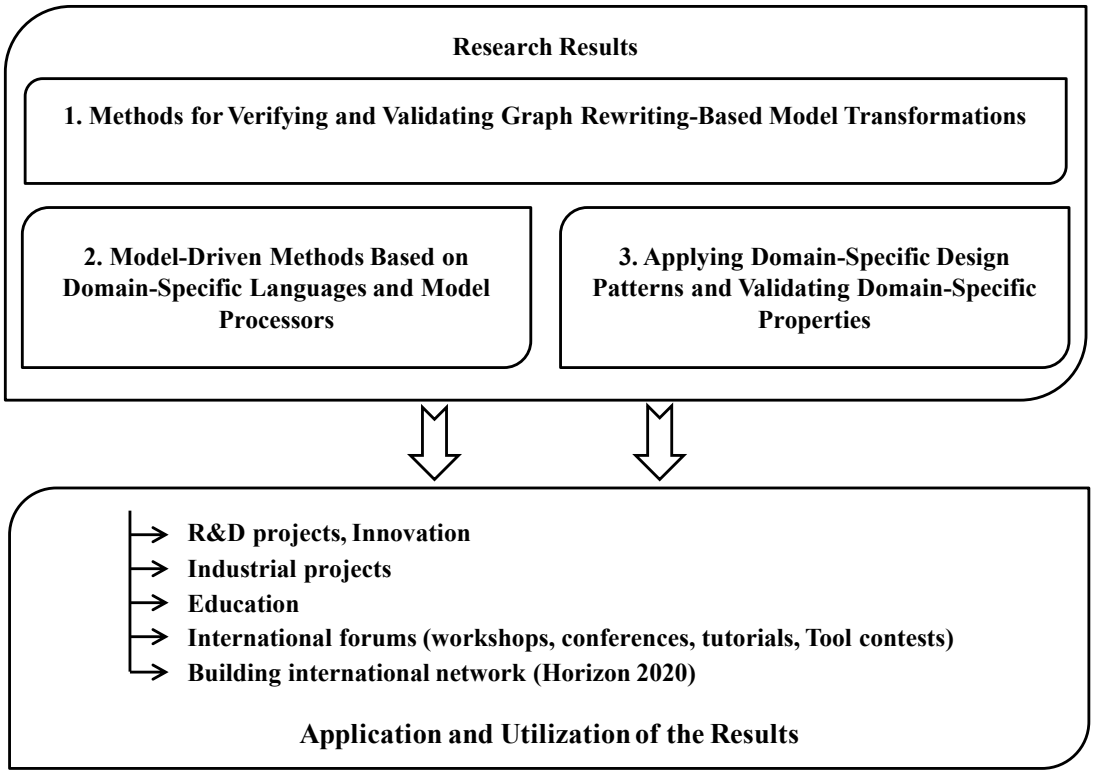


Figure 6-1 Novel scientific results and their application fields

The developed verification and validation methods, furthermore, their supporting solutions allow the development of higher-quality software products. Figure 6-1 summarizes both the research results introduced in the thesis and their application areas.

Based on modeling and model processing techniques, furthermore, utilizing domain-specific capabilities of domain-specific modeling and model processing, we have worked out methods, case studies, developed frameworks, and realized software solutions. Research activities have been motived and the scientific results have been utilized in the following selected projects:

- Metamodel-based model transformation, T-Mobile (2005-2006)
- Service modeling, T-Mobile (2006)
- Developing mobile services and applications with model-driven methods, Mobile Innovation Centre (2007-2009)
- Software as a Service, T-Mobile (2008)
- Software development method and supporting framework, BME Innovation and Knowledge Centre of Information Technology (IT)2 (2008-2009)
- Multiplatform mobile application development, Nokia (2010)
- Model-driven development framework for mobile applications, IBM (2010)
- Modeling and model processing, BME Research University (2010-2012)
- Model-driven application development for multiple mobile platforms, FuturICT (2012-2014)
- Quality assured model-driven requirements engineering and software development, Quattrosoft (2013-2014)
- Model-driven technology to support multi-mobile application development, BME (2013-2014)
- Domain-specific modeling environment supporting IEC61131-3 standard, Infoware (2013-2014)
- Rule engines in backend applications, model-driven solutions to configure software systems, Nexon (2013-2014)
- SensorHUB: a multi-platform IoT Framework, BME AUT (2014-)
- URBMOBI – Urban Mobile Instruments for Environmental Modelling, European Institute of Innovation and Technology (EIT) Climate-KIC (2013-2015)
- SOLSUN – Sustainable Outdoor Lighting & Sensory Urban Networks, Sustainable City Systems, European Institute of Innovation and Technology (EIT) Climate-KIC (2015-2017)

Multi-Paradigm Modeling (MPM) workshop series has a significant role in disseminating the research results at international level. MPM is held annually within the framework of the MODELS conference. MODELS is perhaps the most important annual conference on software modeling and related technologies. The workshop is organized since 2007 with an active and continuous involvement of our research group.

The results of the research activities have been realized in the VMTS framework [VMTS]. The framework has participated several times successfully on international *Tool Contest* competitions [18].

We have a strong and fruitful relationship with the Matlab Simulink R&D engineers. In 2012, we have presented a joint tutorial at MODELS conference. The tutorial introduced how to process Simulink models with VTMS applying graph rewriting-based model transformation [25].

Active participation in both national and international R&D projects, the application of the developed methods, collection of the industrial feedback, furthermore the systematic and iterative processing of

these experiments are always present in our work. Results of the research activities appear in the education: earlier in *Model-driven paradigms* MSc course, currently in both *Distributed systems and domain-specific modeling* and *Software development methods and paradigms* MSc courses. We continuously share our experience related to the education of the software modeling and model processing. Most recently, in 2012, we have summarized our findings and suggestions on international level at the *Educators' Symposium* held in MODELS conference [4]. In 2014, we provided again a tutorial at MODELS conference, the topic of this tutorial was the effective creation of domain-specific languages and the work with them [34].

Based on the common work with the industry, we see that industrial partners need the working solutions. They are about to apply efficient methods in their development and implementation processes. We, as research teams, should support the industrial community with high quality methods and solutions.

The following part of this chapter summarizes the software applications and tools developed within the scope of the research activities, then, discusses few representative research and development projects utilizing the results, finally, summarization is provided.

6.1 Software Applications and Tools Developed within the Scope of the Research Activities

The primary area, where the results have been applied, is the Visual Modeling and Transformation System (VMTS) and the projects based on VMTS. Domain-specific modeling, furthermore, model processing results and knowledge contribute to the VMTS framework and have been utilized in several projects [6]. We have developed various graphical and textual languages covering several domains. The first languages were the various UML dialects: class and object diagrams, activity diagrams, sequence diagrams, use case diagrams, others were followed by languages to implement a variety of domains (for example, bank, insurance, service tree, database, social network, server network, user interface, communication protocols, workflow, feature models, robots, control, model processing languages). Designed and developed model processors also cover a wide range: processing UML and further models, generating source code, supporting the generative programming paradigm [Beuchat et al, 2006] [Czarnecki and Eisenecker, 2000] [Donohoe, 2012] by normalizing and processing feature models, generating source files targeting embedded systems, for example, generating code for Quantum Framework [Samek, 2002] based on state charts, supporting software evolution, furthermore processing various domain-specific models, generating configuration files, traversing and optimizing models.

VMTS framework provides both static and dynamic validation method. The results presented in the thesis provides the dynamic approach.

By using VMTS Animation Framework, we have developed an interactive visual model transformation debugger. The debugger can execute the transformation rules step-by-step [31]. The integration with the Matlab Simulink system allows the animation of the Simulink models within the VMTS. This is summarized in the introduced *Method for Processing Mathworks Simulink Models with Graph Rewriting-Based Model Transformations.*

Application of the scientific results is not strictly limited to VMTS framework and the related projects. For example, the results of the *Assuring the Quality of Software Development Projects by Applying Model-Driven Techniques and Model-Based Tools* area, adapting the needs and the basic tools of the industrial partner, have been realized in the Eclipse environment [Eclipse 2016] as an Eclipse plugin.

Several partners applied and still use modeling and model processing results developed in common projects. E.g. T-Mobile, IBM, Quattrosoft, Nokia Research, Nokia Siemens Networks, Infoware and Nexon.

Beside the VMTS, our second key framework is the SensorHUB [9]. The following sections introduce both the VMTS and SensorHUB frameworks, furthermore their integrated utilization, i.e. the multi-domain IoT approach.

6.1.1 Visual Modeling and Transformation System

Visual Modeling and Model Transformation System (VMTS) is a multipurpose modeling and model processing framework. The framework is introduced in Section 2.9.

6.1.2 SensorHUB Framework

Software systems covering data collection, transmission, data processing, analysis, reporting, and advanced querying are usually developed by strong method and framework background. Consolidated development methods and frameworks provide the efficiency and ensure the quality of the software artifacts. SensorHUB framework utilizes the state of the art open source technologies and provides a unified tool chain for IoT related application and service development. SensorHUB is a platform as a service (PaaS) solution for IoT and data-driven application development. The strength of the framework is that it covers the whole data collection, analyzing and reporting process. SensorHUB provides a unified toolchain for IoT-related application and service development. SensorHUB framework, next to the IoT-related application and service development, supports the data monetization by providing a method to define data views on top of different data sources and analyzed data. [9] [SensorHUB]

SensorHUB makes it possible to develop and reutilize domain-specific software blocks, for example, components of the smart city domain or the vehicle domain that are implemented once and can be built into multiple applications. The framework makes them available by default and provides various features to support developers working in the field.

SensorHUB also provides tools to support application and domain-specific service development. The architecture of the concept is depicted in Figure 6-2. The whole system contains the following areas:

1. Sensors, data collection, local processing, client side visualization, and data transmission (bottom left)
2. Cloud-based backend with big data analysis and management (bottom right)
3. Domain-specific software components (middle)
4. Applications, services, visualization, business intelligence reports, dashboards (top)

Sensors cover different domains: health, smart city, vehicle, production line, weather and further areas. Local processing and data transmission makes up a local platform, which performs core services, i.e. data collection, data aggregation, visualization of raw measurements, secure communication, and data transmission. This component also provides information as a local service interface for different applications.

The cloud component provides historical data storage, big data management, domain-specific data analysis, and extract-transform-load (ETL) mechanisms. Its architecture was designed specifically for cloud deployments, although it can also be deployed on premises. In the core, we have designed a service layer based on the microservice architecture. The loosely coupled services make up an important part of the framework. The most notable domain-agnostic services are the data ingestion service and the general querying service. Among the more domain-specific services are the push notification service, which is applicable in all domains that have smartphones on the client side, and the proximity alert service, which can be used to determine if the sensor is located inside a predefined area and is useful in the transportation or agricultural domains.

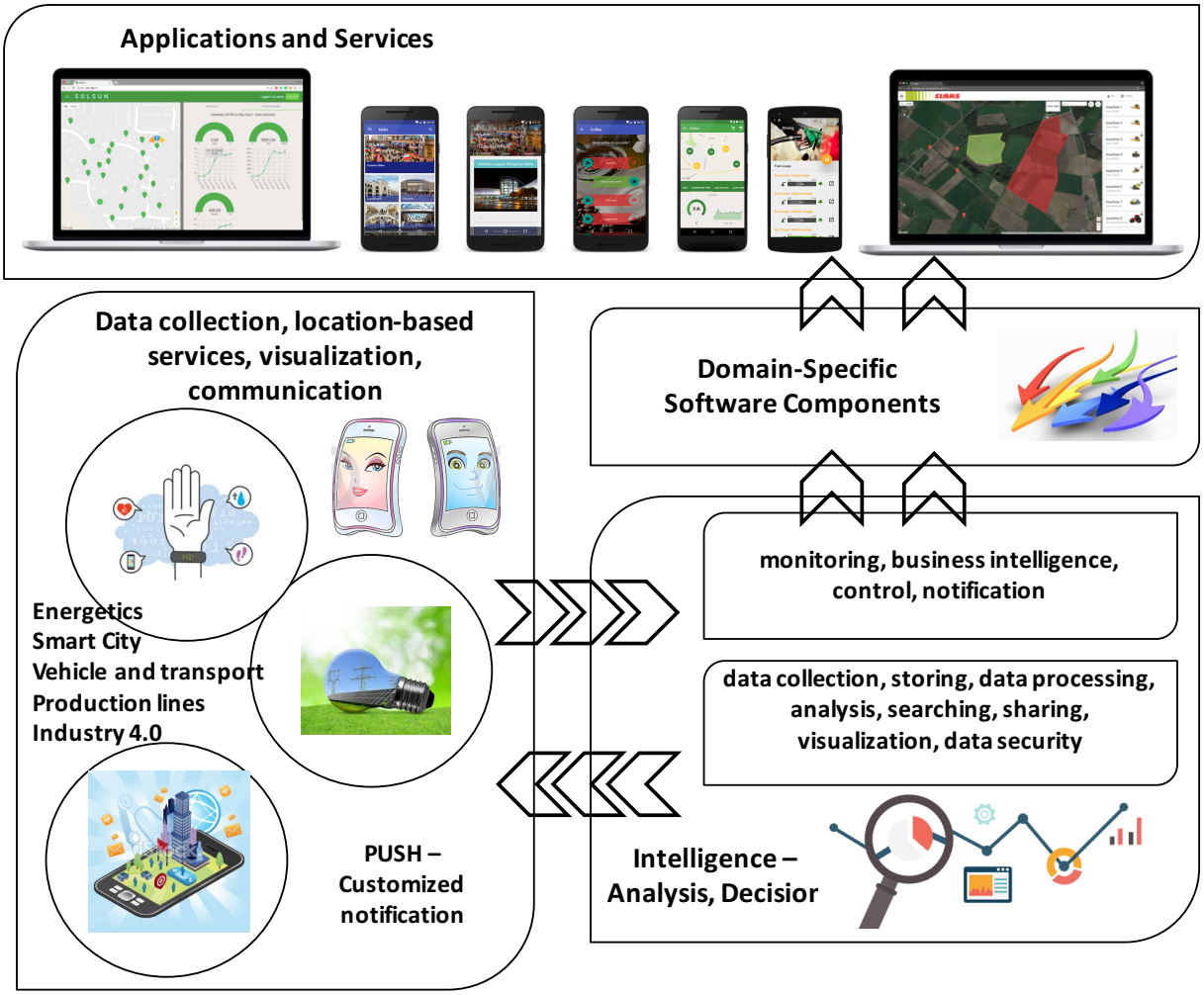


Figure 6-2 Architecture of the SensorHUB

The last layer comprises applications that implement specific user-facing functionalities. These data-driven applications, independently of their purpose, eventually face the same problems repeatedly. Without framework support, applications should find a way to collect data, to store large amounts of data reliably and in a scalable way, to transform data into a format convenient for data analysis, or present data on a dashboard. Solving these problems is not trivial, and can account for the majority of the development effort if done one-by-one for every different application. The main purpose of the SensorHUB framework is to function as a platform for these applications, providing the implementation of the previously described areas, so the application developers can focus on the domain-specific issues they intend to solve.

SensorHUB framework is developed in an incremental and iterative way, the loose coupling between its modules makes us able to develop and update components independently.

We utilize the following technologies and components during the implementation of SensorHUB:

- *Node.js* is applied as a cross-platform runtime environment for server-side applications. It provides an event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. [Node.js]
- *Docker* is an open-source software container framework. It provides virtual machine-like separated environments with very little overhead. We used it for packaging and deploying the microservices. [Docker]

- *MQTT* is a lightweight messaging protocol based on the publish-subscribe model working on the top of the TCP/IP layer. We use an implementation of this protocol for direct two-way communication with different sensors and devices. [MQTT]
- *Apache Kafka* is applied as a central hub performing load balancing, queuing and buffering incoming data from various sources including MQTT brokers and the microservice layer. Kafka is able to process thousands of incoming data packets per second, making it a perfect choice for this task. [Apache Kafka]
- *Apache Hadoop* is used as a software framework for distributed storage and processing of large data sets on computer clusters built from commodity hardware. It consists of a distributed file system (HDFS) and a resource management platform (YARN), furthermore, it provides a basis for a great deal of purpose-built frameworks, such as Ap*ache Spark, Apache Hive* and *Cloudera Impala*. [Apache Hadoop]
- *Apache Spark* is a high-performance cluster computing framework. We utilize the high-level functional API of Spark for data processing and Spark Streaming for effective real-time event-processing. [Apache Spark]
- *Apache Hive* is applied as a data warehouse infrastructure built on top of Hadoop. It provides an SQL interface for data stored on HDFS. We use it for ETL (extract, transform, load) batches, which require high throughput instead of low latency. We also utilize *Cloudera Impala* for this purpose, as it provides faster queries. [Apache Hive]
- *Apache Cassandra* is a distributed, massively scalable NoSQL database. SensorHUB applications can use this form of data storage for quick queries against processed data. Cassandra is capable of ingesting tremendous amounts of data very quickly; this makes it a great choice for large-scale IoT applications. [Apache Cassandra]

6.1.2.1 Detailed Framework Architecture

Given the scale the framework needs to operate on, we designed it to be deployed in cluster environments (clouds). We have organized the different functionalities into microservices. Microservices are light-weight server components that focus on a single task. This approach not only makes the services more maintainable, easier to develop independently of each other and replaceable, but also leads to components that boot fast, which is an essential requirement when deploying to the cloud, as new instances must be fired up on the run as the load increases. Most of the framework's microservices are built using the Node.js framework or Java technology, because they are light-weight and excels at I/O-heavy tasks.

We made the different microservices accessible for applications through an API Gateway, which unites the microservices into a cohesive interface and hides all the service instantiation, discovery and load balancing details from the applications. Further service of the API Gateway is to authenticate applications before using the framework. Load balancing and authentication is based on the microservice repository and the application repository. These two services, running and tracing service instances, and registered client applications serve as the backbone of the framework.

The microservices are deployed in separate Docker containers. Docker is supported by all major cloud providers and using this technology makes distributing and managing the framework seamless. The clustered running and scaling of the components can be orchestrated with tools such as Kubernetes or Docker Swarm. Based on the measurements, booting up a Node.js instance is relatively quick, compared to a Java-based solution, furthermore, by keeping the services stateless, the load balancing task is

straightforward in any environment. Figure 6-3 provides a detailed overview of the framework architecture.

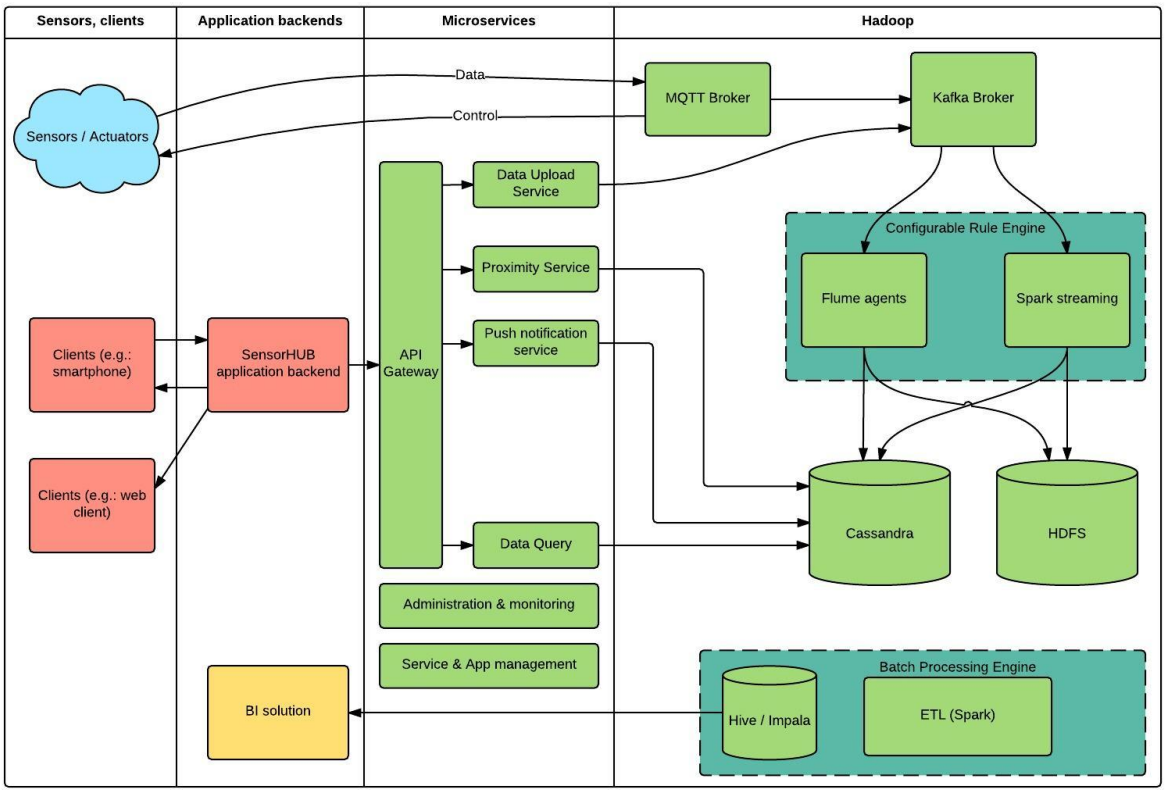


Figure 6-3 The detailed architecture of the SensorHUB framework

Data ingestion and data querying microservices are the two pillars. Data is uploaded into a cluster of machines running Hadoop by the Data Upload Service. Raw data can be queried using the Data Query Service.

The framework also provides an MQTT-based way for data upload. In certain cases, sensors, actuators or any client device can directly communicate with the platform, without the overhead of an application backend and layers of microservices. For these use cases, we provide the MQTT-based endpoint. Using this endpoint, the platform can receive data on large scales and is also capable of sending back control or configuration instructions.

The MQTT and microservice-based ingestion methods load data into an Apache Kafka cluster, which is the entry point for the Hadoop platform.

Providing a schema for the ingested data is not required. The schema is forced on the raw data by the application itself. This method gives great flexibility, however, in certain cases, having a fixed schema provides benefits, i.e. automatic code or job generation can be performed based on schema information. Therefore, the framework allows to store schema for a given dataset or data source. A further advantage of this hybrid approach is a standard query interface for the data that has a provided schema. Applications, which do not support metadata, handle the query interface themselves. This is a reasonable tradeoff between customizability and the ability to use general services provided by the framework.

6.1.2.2 Data Processing

Although flexibility is an asset, in most cases the schema is known at the time of data ingestion. This is the reason, why we apply a hybrid approach by providing an ETL engine. In this way, application developers can configure loading their data into one of the supported query-optimized data stores. Depending on the needs of the application, the data store can be one of the followings (Figure 6-4):

- HDFS file system as a Data Lake (a large-scale storage repository and processing engine): this storage should never be modified and should serve as a secure and reliable historical data storage for further processing or archive purposes.
- A compressed, partitioned, columnar data store, implemented on a massively parallel processing engine (such as Apache Hive or Cloudera Impala with Parquet files) that is efficient for analytic query patterns.
- A NoSQL data store (Apache Cassandra) that is utilized for fast data retrieval, data modification and simple analytic queries, e.g. queries from client applications displaying live or historic data to end-users.
- A traditional relational database (MySQL) with the advantage that it is well known to developers has several associated tools and scales well for medium-sized services.
- Data can be piped into a stream processing module, which can be used to detect anomalies in the incoming events and send immediate alert messages directly to client applications or do any logic required by the application.

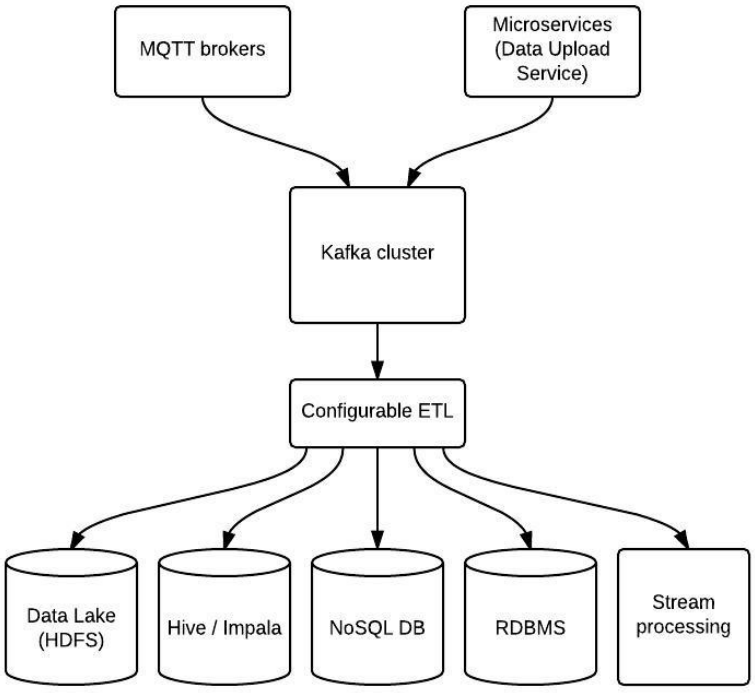


Figure 6-4 SensorHUB data store variations

In scenarios, where raw data is not necessarily stored, but is just processed in a streaming-like fashion, using the Data Lake is optional, but recommended. As data in the Data Lake is never modified once uploaded, application developers can always access data with arbitrarily complex processing algorithms or by providing their own custom ETLs. These standard formats, supplemented by the capability of

defining further custom processing algorithms, enable developers to focus data at the abstraction level that best fits their needs, contributing to the ease of development.

6.1.2.3 Deployment

On top of the platform, there are the domain-specific web and mobile applications and services. Special types of services are customized reports, data monitoring solutions, dashboards, and further business intelligence solutions. As the platform itself is designed to be deployed on a backend infrastructure of an internal network, it is recommended that these applications use their own separate servers to utilize the capabilities of the platform. It is also possible to simply open the internal ports to client applications, but this is not advised, because it would introduce security risks. Internal microservices are prepared to authorize requests that are coming from a relatively safe, firewall-protected environment, not from the outside world. In the current architecture, application of these strong security measures is the responsibility of the application-specific web servers.

Figure 6-5 shows a possible setup, where the Hadoop Cluster and the SensorHUB platform are deployed on internal network servers, and the different user-facing services deploy their own web servers. An example would be an application that uses smartphones to collect data and provides services to the users. Such smartphone applications would directly connect to their own backend servers, knowing nothing about the SensorHUB framework. The application backend would wrap the services of the underlying framework, and glue them together in a way best fitting for the application. One of the main strengths of the SensorHUB framework is that it enables the application backend to remain a thin layer. In the absence of the framework, every single application would need to implement its own version of the data handling functionalities.

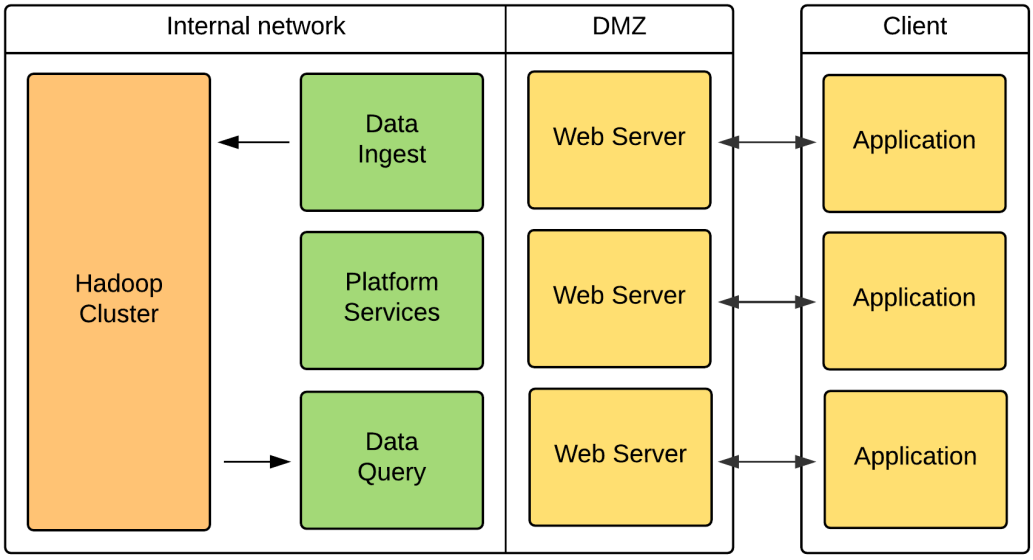


Figure 6-5 A possible deployment of the SensorHUB framework with client applications

In many of our SensorHUB utilizations, a smartphone running Android OS serves as a bridge between a sensor and the infrastructure in the cloud. As many of these sensors have no direct internet access, but are capable of communicating using Bluetooth or Wi-Fi, an Android smartphone with the capability of Bluetooth/Wi-Fi connection and mobile internet access is ideal for this purpose.

6.1.2.4 *Client-side Support*

In order to support the client application development, we provide client-side services. They are implemented on the Android smartphone platform and distributed as an application library. This library encapsulates the client-related services, and provides them as independent building blocks.

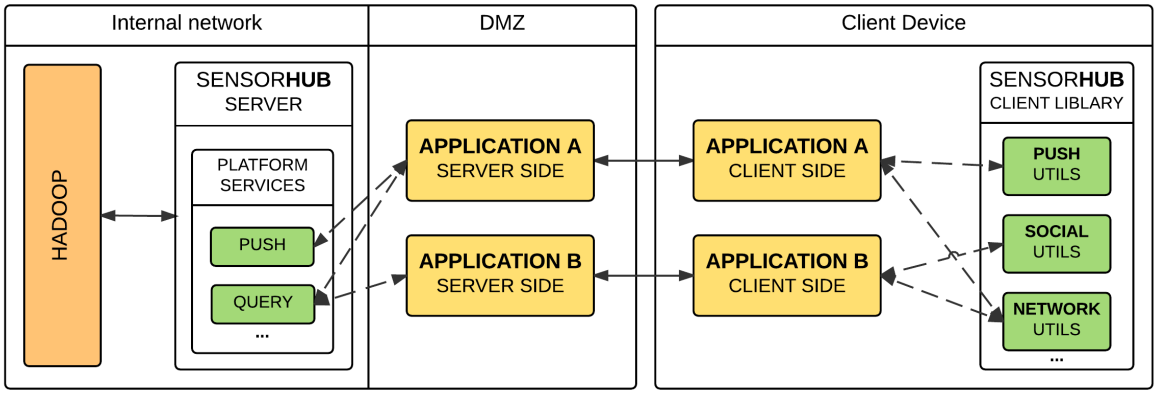


Figure 6-6 The environment of an application that utilizes the SensorHUB framework

The first part of these modules are client-side counterparts of the platform services available on the infrastructure side. These are client-side utilities that support client services, such as transparent push notification handling and device registration, or data querying.

The second part of the client modules are the utilities. These modules provide services for common client-side domain-independent features, including reliable networking, secure communication, and integration with social services (Figure 6-6).

6.1.2.5 *SensorHUB Summary*

SensorHUB is a general concept with a core platform implementation. We provide different realizations (domain-specific software components), i.e. utilizations of the SensorHUB platform. The results are different specialized platforms targeting a selected area.

The framework has been successfully applied as a development accelerator framework for the smart city, vehicle, health, and production line domains. Several SensorHUB-driven systems, targeting smart city, agriculture and health areas are also under design and development. Some of them are introduced in the next sections of this chapter.

6.1.3 Multi-domain IoT

This section introduces the ***Model-driven Multi-Domain IoT*** concept. In a multi-domain IoT environment, data comes from several sources: sensors that collect traffic, health, climate and further information, posts on social networking sites, digital images and videos, records of purchase transactions or mobile phone GPS signals to name some of the most significant. [9] [40]

We see Multi-Domain IoT as the actual frontier for innovation, competition, and productivity. The introduced method supports effective service and application development and therefore covers the following areas: connected devices (connectivity, intelligence), data collection (sensors, storage), data access (cloud, standards, open APIs, security), complex analytics (big data tools), and unique value (realization of the true potential driven by the connected society).

The key points of the worked out method and tool ecosystem are:

- **Step 1.** ***IoT (Measuring, Thinking and Doing).*** IoT is about to increase the connectedness of people and things. IoT ecosystems can help consumers achieve goals by greatly improving their decision-making capabilities via the augmented intelligence based on the collected and analyzed data.
- **Step 2.** ***Multi-Domain IoT (SensorHUB).*** SensorHUB is both a method and a framework to support IoT-related application and service development. Furthermore, it effectively supports the discovery of data correlations that drives the product improvement, service development and the efficiency of the business activities.
- **Step 3.** ***Model-driven Multi-Domain IoT (VMTS + SensorHUB).*** The utilization of software modeling and model processing techniques is provided to enrich the service and application development for the IoT area and improve its efficiency. As a result, we can increase both the development productivity and the quality of software artifacts; furthermore, we can significantly reduce the time-to-market.

Our team has developed both the SensorHUB and the Visual Modeling and Transformation System frameworks. SensorHUB focuses on the IoT-enabled service and application development, including the multi-domain support, while VMTS is our software modeling and model-processing environment. The ***Model-driven Multi-Domain IoT*** concept utilizes the capabilities of both of them; furthermore, it realizes model-driven, quality assured service and application development for the Multi-Domain IoT area. The efficiency of the method is confirmed by several projects. Selected parts of these projects are discussed in this thesis, in order to support understanding and serve utilization of the method.

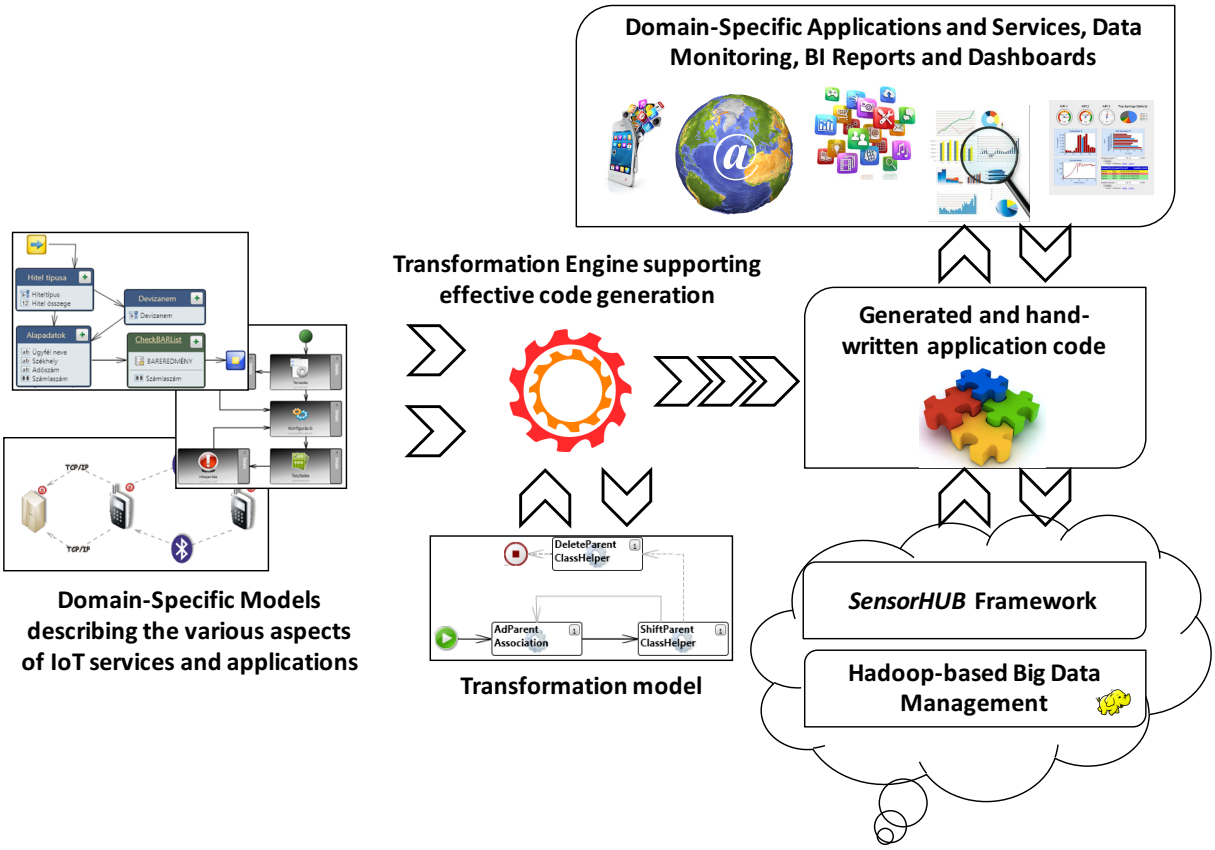


Figure 6-7 Overview of the *Model-driven Multi-Domain IoT*

To achieve the goals of the ***Model-driven Multi-Domain IoT*** concept, we apply model-based methods. The architecture of the application generation process is depicted in Figure 6-7. Using different domain-

specific models, we describe various aspects of the IoT-based services and applications. We describe the data collection interface as the configuration of the backend system, define the data processing rules, push notification rules, data query capabilities, query parameters, service interfaces, application features, and further aspects of the envisioned IoT system. With a help of the VMTS framework, we can manage these languages and the domain models. We also apply domain-specific model transformations to process system models and generate source code snippets and software components from the domain models. Custom business logic can also manually added to the generated code and thus a complete source code package is created. This package relies on the SensorHUB framework, i.e. the code utilizes the framework APIs, class libraries and various services. This way, the generated code acts as a layer on top of the framework, so that we can reuse the previously prepared, well-tested and effective components of the SensorHUB framework. Usually, applications require custom design that is manually prepared and applied for the web and mobile user interface components.

Regarding the SensorHUB framework, the role of domain modeling is to reduce the amount of repetitive coding tasks, decrease the time to market when introducing a new component, or changing the configuration of the framework and to increase clarity of the solutions used. At first, we have identified steps, which can be aided by domain modeling and model-based solutions. We have chosen three domains: (i) configuration files for Apache Flume, (ii) interface and data structure specification for clients and for the server side based on Swagger, (iii) filter configuration for Big Data processing. While creating environments for these domains, we consulted continuously with domain experts and fine-tuned our solutions according to their suggestions.

6.1.3.1 Configuration for Apache Flume

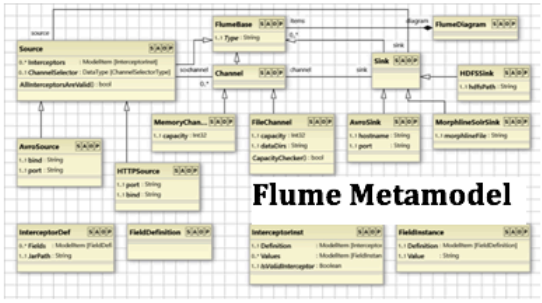
Flume is a distributed and reliable service for efficient collecting, aggregating, and moving large amounts of log data. Flume has a simple and flexible architecture based on streaming data flows [Apache Flume]. In our solution, we have introduced a simplified graphical interface to specify Flume configuration files earlier created manually. As a result, we obtain models, which are more compact, easier to understand and faster to edit than the configuration files.

Based on the Flume specification [Apache Flume], we have created a simple, graphical DSL consisting of *Sources*, *Channels* and *Sinks*. We have also defined subtypes of these elements, where each type has attributes as defined in the specification. For example, we have an *AvroSource* (subtype of *Source*) with attributes *bind* and *port*. Note that for the sake of clarity and simplicity, only those types and those attributes were added to our language, which are used by the programmers. The language definition can easily be extended later without losing previously created models.

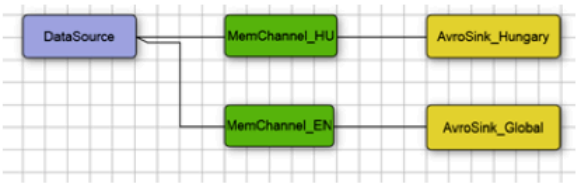
When the basic language was completed and tested, we have also added useful advanced features, such as multiplexing flows with selectors and Interceptors. Realizing Interceptors was challenging, since users can create an Interceptor definition (list of fields) and then use and fill out this definition (using the fields as variable slots) in Sources. E.g., a definition has two fields: *MimeType* and *Encoding*. Then, in a *HTTPSource*, we can add an Interceptor using this definition and set the concrete values as *MimeType= text/html, Encoding= UTF8*. This feature requires dynamic instance-level management of Interceptor fields, for example, in case the definition is modified. We have realized it as the part of the domain-specific modeling add-on in VMTS IDE.

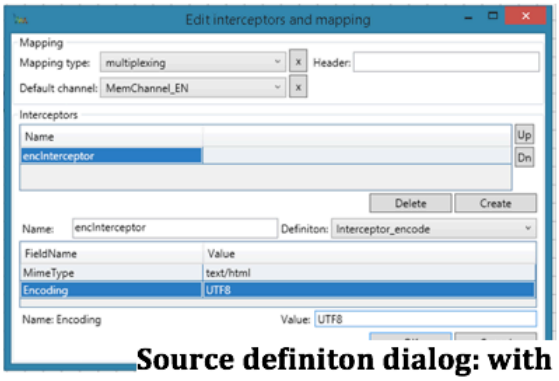
We have created several example models, then extended the validation logic and finally added a code generator as a part of the modeling add-on. The code generation is rather simple in this case: we have to traverse the models and for each model element generate an appropriate set of code lines. The code generator mostly consists of nested iterations listing the attributes of model elements.

The language has been tested and the solution has been verified by the developers. As a result, it become clearer and faster to apply changes in this domain environment than it was before by manual coding.



Flume Model in the graphical editor





Source definiton dialog: with mapping & interceptor

```
# List the sources, sinks and channels for the agent
FlumeDiagram.sources =  DataSource
FlumeDiagram.channels =  MemChannel_HU MemChannel_EN
FlumeDiagram.sinks =  AvroSink_Hungary AvroSink_Global
...
# Properties of the objects
# The properties of the source DataSource
FlumeDiagram.source.DataSource.bind = 152.66.178.34
FlumeDiagram.source.DataSource.port = 7075
FlumeDiagram.source.DataSource.Type = AvroSource
...
# Interceptors
FlumeDiagram.sources.DataSource.interceptors =  encInterceptor
FlumeDiagram.sources.DataSource.interceptors.encInterceptor.type = Interceptor_encode
FlumeDiagram.sources.DataSource.interceptors.encInterceptor.MimeType = text/html
FlumeDiagram.sources.DataSource.interceptors.encInterceptor.Encoding = UTF8

# Bind the source and sink to the channel
FlumeDiagram.sources.DataSource.channels
...
```

Generated Flume config file

Figure 6-8 Model processing

6.1.3.2 Interface Definition by Swagger

Swagger is a RESTful API representation [Swagger]. Swagger definitions are used to generate interfaces for client applications and for the server-side services.

Swagger definitions can describe method calls with parameters, return types and error codes. All three parts have a type, which can be either one of the built-in primitive types, or a custom type consisting one, or more fields, e.g. a custom type *Person* with *FirstName* and *LastName* fields. Types may have additional attributes, such as *Format*, which acts as a constraint on the type (e.g. integers under 1000). Earlier, Swagger definitions were written manually, which was error-prone, since the precise usage of whitespace characters was required (e.g. the length of a tab was specified). Furthermore, there has been no autocomplete feature in the editor; thus, the spelling of type names was also a common source of errors.

We created a DSL to describe the types and the hierarchical relations between them. We have also created a built-in read-only model describing the built-in types, thus users can use (refer to) them automatically in their work. Based on this language, we built a domain environment. The environment consists of dialogs, where the selected type components can be edited in a table editor. Since type names are chosen from a dropdown list, spelling errors are automatically eliminated. Furthermore, the dialogs can validate *Format* descriptions when the users close the dialogs. The language and the environment were exhaustively tested. We can state that they are able of describing arbitrary type systems used in practical case studies.

The second step in creating the model-based solution was to extend the language with method call definitions. It was fairly simple, since method call definitions are rather similar to composite types (even

if they have parameters instead of fields), but we had to add several descriptors to the definitions as well (e.g. post/get method, URL). We reused and extended our previous language in less than one day of work.

By completing the language, we moved the focus to the code generator. As in the case of the Flume domain, this step was simple: we had to traverse the model trees and generate the Swagger text by applying the *Visitor* design pattern.

6.1.3.3 Filter Configuration for Big Data

In the third domain, the basic motivation is to process the data of the sensors. The data has a large amount of information fragments; we would like to understand these fragments, combine them and obtain answers to several questions asked by the users. While processing the fragments, we often transform them and filter out important parts while omitting everything else. For each question, we can specify the series of data manipulation tasks referred to as filters, which produce the answer. This process can be modeled quite well. Each filter has an interface definition, i.e. input and output parameters. We created a graphical language, where filters are represented by boxes. Filter definitions are managed similarly to type management in case of Swagger and boxes have input and output ports representing the parameters of filters. Boxes are connected to each other through these ports. Input and output data of the whole configuration is represented as the interface of the model itself. As the result, we have a workflow language for filters and the filter configuration can be easily described in a user-friendly way.

Code generation is also possible based on the graphical models; however, it is more complex comparing to the previous two domains. For example, we have to obtain the correct sequence of execution from the model. Fortunately, this can be applied by searching for a random filter in the model with no unprocessed input and processing it. If we cannot find such a filter and there are still unprocessed filters, then the model is invalid.

6.1.3.4 Summary of the Model-driven Multi-Domain IoT

The ***Model-driven Multi-Domain IoT*** concept utilizes the advantages of the model-driven system development. Based on the advantages of the VMTS and SensorHUB frameworks, the ***Model-driven Multi-Domain IoT*** provides a unified toolchain for IoT-related application and service development.

We have discussed the motivation, the objectives, the application areas and the domains of SensorHUB framework and its extension to the ***Model-driven Multi-Domain IoT*** concept. Based on present industrial trends, requirements, and needs, a SensorHUB-based method and framework is a data monetization enabler. The framework supports the collection of various sensor data, enables the processing and analysis of data, and makes it possible to define different views on top of the data combined and compiled from different data sources. These data views and collections of datasets are referred to as monetized data for various purposes, for example, supporting decision making and running smart city services.

6.2 Research and Development Projects Utilizing the Results

The following sections introduce research and development projects. These projects have utilized several results introduced in this thesis. Beside the research activities, in each of the following projects I played significant role both in project management and in research and development directions as well.

6.2.1 Modeling and Model Processing

Some of the results have been elaborated within the framework of the R&D projects such as *Developing mobile services and applications with model-driven methods,* Mobile Innovation Centre (2007-2009);

Software Development Method and Supporting Framework, BME Innovation and Knowledge Centre $(IT)^2$ (2008-2009); *Modeling and model processing*, BME Research University (2010-2012); and *Model-driven application development for multiple mobile platforms*, FuturICT.hu (2012-2014). The four projects together cover a significant time interval. In each project, our research group had a determining role. We achieved significant values in the field of domain-specific modeling and model processing areas. We elaborated these results in VMTS, and they have been utilized during several system design and application development activities, such as certain elements of the professional modeling and model processing are utilized in the SensorHUB framework [9].

6.2.2 Quality Assured Model-Driven Requirements Engineering and Software Development

Software development requires appropriate methods for requirements engineering, design, development, testing, and maintenance. The more complex the system is, the more sophisticated methods should be applied. A significant part of software projects is short on appropriate processes. This project has developed a quality assured model-driven requirements engineering and software development method. The method is based on the modeling of the software requirements in a way that these models can be used to automatically generate several artifacts during the engineering process. This method is continuously developed during the last twelve years driven by our software projects and by the experience and lessons learned from these projects. In the 2013-2014 period, a new tool support has been developed. The method is a framework to specify software requirements with four domain-specific languages and automated methods to process the models. The project also collected and shared our best practices on the field of model-driven requirements engineering and software development. Based on the feedback from research groups and industrial entities, both the method and our experience-based recommendations can be widely applied by other teams and projects. [8]

6.2.3 Model-Driven Technology to Support Multi-Mobile Application Development

Mobile devices and mobile applications have a significant effect on the present and on the future of the software industry. The diversity of mobile platforms necessitates the development of the same mobile application for all major mobile platforms, which requires considerable development effort. Mobile application developers are multiplatform developers, but they prioritize the platforms, therefore, not all platforms are equally important for them. Appropriate methods, processes and tools are required to support the development in order to achieve better productivity. The main motivation of our research activity was to provide a method, which increases the development productivity and the quality of the applications and reduces the time to market. We have provided a mobile, platform-independent, high-abstraction level environment for mobile application design. We support it with domain-specific languages and effective model processing solution.

The project provided significant model-driven results on the field of multi-mobile platform development. Then, we continued the work to support mobile application developers, i.e. to extend the capabilities of the introduced approach and framework by covering more areas of mobile applications. [6]

6.2.4 Supporting Human Resource Management Frameworks with Rule Engine-Based Solutions

The *Human Resource (HR) rule engine calculations* project targeted the HR domain, especially the rule-based configuration of the work schedule and salary calculation algorithms and methods. The goals of the project were to provide an efficient way to configure standalone installations of a HR framework. These installations are based on a common source, they are customized according to country related laws and company-based requirements. The motivation of the model-based solution was to support

several rule environments (area-based, company-based, law-based rules) in a transparent way; provide efficient change management; filter out development errors; focus on domain solutions, not on coding techniques; and fit into the existing architecture.

We have worked out two domain-specific languages and the supporting model processors. The languages are the *System level* language and the *Rule level* language. On the system level, the solution provides a graphical language for high abstraction level overview. This allows to define rules, functions, data structures, and dependencies between rules. The rule level is designed to support calculation algorithms with a complex, detailed but compact textual language.

With the project, we realized a more efficient change management process, provided support for customization but kept the unified representation, models could be validated, code quality have been improved. Furthermore, the solution is extendable on different levels (rules, data structures, built-in functions), could be adopted to existing systems and the performance of the processing has been significantly improved.

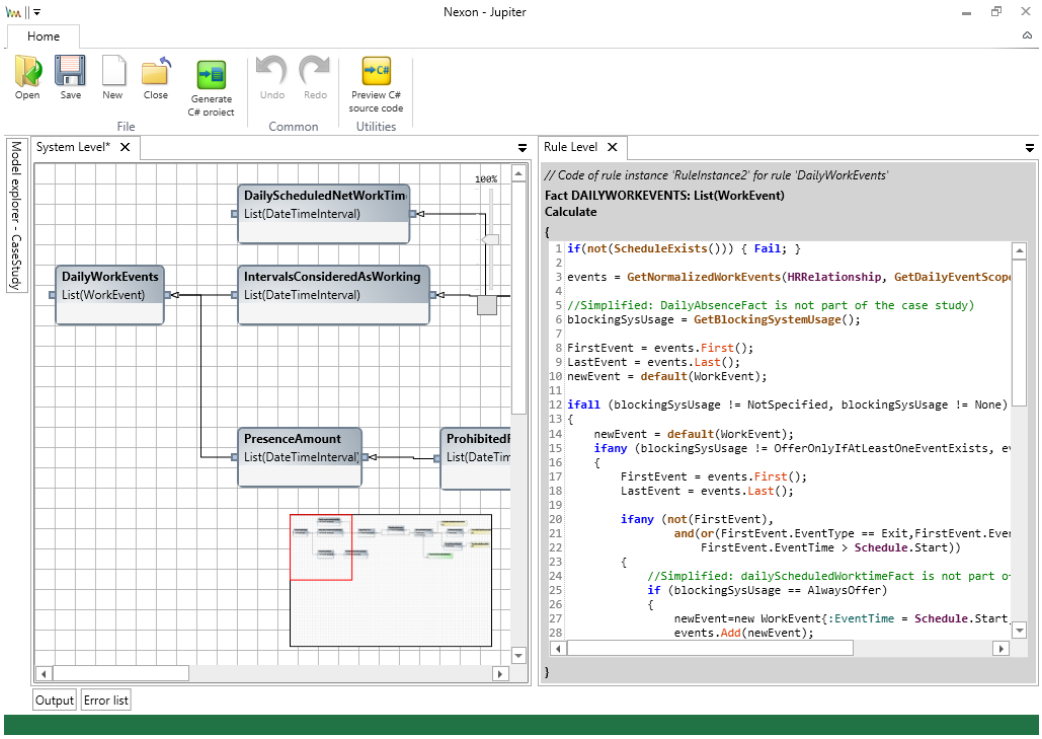


Figure 6-9 HR Rule Engine user interface in VMTS

6.2.5 Graf IEC

IEC 61131 is an industry standard for PLCs. The standard defines graphical and visual languages, and rich sets of built-in functions and program function blocks. Several parts of the graphical and textual languages are interchangeable.

The Graf IEC project provided a VMTS-based modeling environment with both textual and visual domain-specific languages based on the IEC 61131 standard and further custom requirements: custom built-in functions (e.g. for debug); different semantics for variables; code generation for C; code generation for a custom macro language; ability of embedding resource files; simulation; and support for domain-specific design patterns.

The worked out software solution has addressed these requirements, provided a workbench for the IEC 61131 standard, supports code and documentation generation, model validation, and software-based simulation as well.

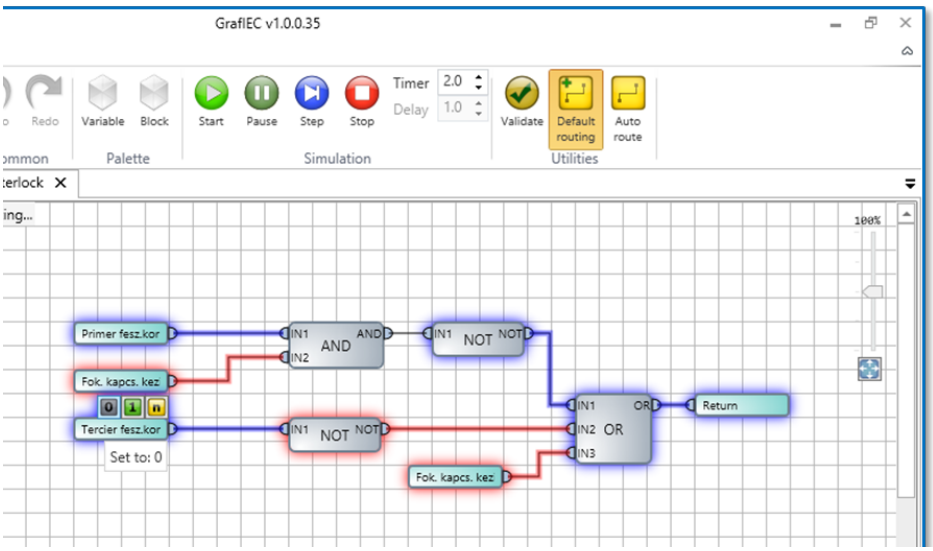


Figure 6-10 Graf IEC user interface in VMTS

6.2.6 Several Domains, Big Data, Big Challenges, Great Opportunities

Challenges and opportunities of the IoT and big data areas include analysis, capture, search, sharing, storage, transfer, and visualization of data and information. Management of the collected data and the attendant security concerns are among the biggest challenges. *What does data mean?* – This is a key point we often face. However, we believe that *big data* and the *IoT world* allow customers to get beyond reactive and even beyond proactive, to become predictive. We can take a more holistic view of the tools and their behavior.

Combining the experience and results from previous projects targeting various IoT domains, a configurable set of general modules has emerged, which we call SensorHUB. The concept continuously evolves, based on feedback from R&D and industrial projects.

VehicleICT platform is an implementation on top of the SensorHUB framework targeting the vehicle domain. The implementation of the VehicleICT platform helped to distill the architecture of the SensorHUB. VehicleICT utilizes the capabilities of the SensorHUB and provides a vehicle domain related layer with several reusable components and features. This means that VehicleICT platform itself can be considered as a test environment that verifies the different aspects of the SensorHUB framework.

The idea behind the VehicleICT platform was to identify a reasonably rich set of functionalities that typical connected car applications need and then to implement and test these functionalities and finally offer them as building blocks in a centralized manner. VehicleICT was one of the first projects, where both the client and server parts of the SensorHUB framework have been utilized [Lengyel et al, 2015] [VehicleICT].

We worked out the concept of our *Social Driving* solution, where the goal was to motivate and help car owners to drive more efficiently. *Social Driving* application is based on VehicleICT platform. *Social Driving* shows statistics about driving style, fuel consumption and CO_2 emission. The solution runs in the background, therefore, it does not interfere with other mobile applications. The application uses the sensor data both from the OBD and the mobile phone. [Ekler et al, 2015]

6.2.6.1 Smart City Domain

Within the frame of two EIT (European Institute of Innovation & Technology) Climate-KIC [EIT Climate-KIC] projects, we utilize the framework. These Climate-KIC projects are referred to as URBMOBI (Urban Mobile Instruments for Environmental Monitoring, i.e., a Mobile Measurement Device for Urban Environmental Monitoring) [URBMOBI] and SOLSUN (Sustainable Outdoor Lighting & Sensory Urban Networks) [SOLSUN].

The *URBMOBI* (Urban Mobile Instruments for Environmental Monitoring, i.e. a Mobile Measurement Device for Urban Environmental Monitoring) project integrates a mobile measurement unit for operation on vehicles in urban areas (i.e. local buses and trams), with data post-processing, inclusion in enhanced environmental models and visualization techniques for climate related services, environmental monitoring, planning and research needs.

URBMOBI is a mobile environmental sensor that (i) provides temporally and spatially distributed environmental data, (ii) fulfills the need for monitoring at various places without the costs for a large number of fixed measurement stations, (iii) integrates small and precise sensors in a system that can be operated on buses, trams or other vehicles, (iv) focusses on urban heat and thermal comfort, and (v) aims at providing climate services and integration with real-time climate models.

The URBMOBI solution provides a novel product that integrates state-of-the-art sensors for environmental variables embedded in a system that allows mobile usage and data handling based on geo-location technology and data transmission by telecommunication networks. Sensors can be operated on buses, trams, taxis or similar vehicles in urban areas.

The data is geo-coded and post-processed depending on the type of variable, location and application. Furthermore, the data is integrated into real-time models on climate and/or air quality relevant quantities providing climate services and environmental data for a wide range of applications.

URBMOBI is utilizing the *SensorHUB* framework in data collection, local processing (data aggregation), and data transmission. On the server side, URBMOBI measurements are combined with atmospheric models in order to improve spatial coverage and calculate additional parameters (thermal comfort). The data is analyzed with a climate domain related powerful tool. A part of the *SensorHUB* architecture has been redesigned and improved based on the experience collected at the URBMOBI project. As a result, we have obtained a clearer framework architecture.

URBMOBI project has been worked out between 2013 and 2015 by the following consortium: RWTH Aachen University (Germany), Netherlands Organisation for Applied Scientific Research TNO (Netherlands), ARIA Technologies (France), Budapest University of Technology and Economics (Hungary), MEEO S.r.l - Meteorological and Environmental Earth Observation (Italy), and Aacener Straßenbahn und Energieversorgungsbetrieb (Germany).

The *SOLSUN* (Sustainable Outdoor Lighting & Sensory Urban Networks) project is about to demonstrate how intelligent city infrastructure can be created in a cost-effective and sustainable way by re-using existing street lighting as the communications backbone. We apply different technologies and methods to reduce energy consumption at the same time as turning streetlights into nodes on a scalable network that is also expandable for other applications. Sensors capture data on air pollution, noise pollution and traffic density; information gathered are used to address traffic congestion, another key contributor of greenhouse gas emissions in cities.

SOLSUN project develops an integrated technology platform where both several components of the *SensorHUB* framework and the knowledge of the *SensorHUB* team are utilized. The project brings together a strong core of public, private and academic partners with the combined expertise to develop

outcomes that can be exploited on a global scale. The project is carried out between 2015 and 2017 by the following partners: Select Innovations Limited (UK), British Telecommunications Plc (UK), Municipality of the City of Budapest (Hungary), PANNON Pro Innovation Services Ltd (Hungary), and Budapest University of Technology and Economics (Hungary).

Sensor and sensor network development is performed by Select Innovations Limited, the data collection, storage, analysis and data-driven applications are mainly carried out on SensorHUB architecture.

According to the predictions, up to 100 billion devices will be connected to the Internet by 2020. The SOLSUN technology is designed to be scalable to cope with the growing demand for networked devices. The system can cater for 254 device types with 65,000 devices in one category; multiple protocols are embraced with data sent back to a scalable cloud based Cluster Controller, with no upper limit on the amount of Cluster Controllers. This enables providers to carry on using their preferred protocol but still benefit from a web-based front end and/or application connection. To ensure scalability, connections are made through stand-alone adapters; multiple adapters can be distributed and software can run on many servers with no single point of redundancy.

6.2.6.2 Healthcare Domain

We have seen the emerging popularity of a phenomenon called „quantified self". Followers of this movement regard every aspect of their life as input data, which they record and store in order to improve daily functioning. The history of self-tracking using wearable sensors in combination with wearable computing and wireless communication already exists for many years, and also appeared, in the form of sousveillance back in the 1970s [Swan 2013]. Today, healthcare sensors and different kinds of sport trackers become cheaper and affordable, and even smart devices have sensors capable of performing health related measurements.

The average user collecting self-tracking data is not a medical expert; it is difficult for him to interpret his medical results or similar self-monitoring data in depth. The user is not aware of the importance of the individual values or the meaning of deviance from normal intervals, nor can he combine different measured values to infer his health status. What such users can do is paying for the doctor's time or look up some uncontrolled source on the Internet to learn the meaning of these data.

Motivated by increasing healthcare costs, using medical grade sensors is also regarded as a way of cost-effectively observing the required biological signals of a patient [Pantelopoulos and Bourbakis, 2010]. This phenomenon transforms the healthcare industry in a form where remote experts decide, for example, on the necessity of a surgical intervention for a given patient, based on sensor data collected for days. Similar to knowledge engineering, it is possible to run learning algorithms on voluntary provided sensor data of thousands of users to infer hidden correlations. Automated processes can even warn the user if some suspicious results make it legitimate to visit a general practitioner or a specialist [Clifton et al, 2014]. The experts can harness the availability of historical data during analysis.

A shortcoming of the current state-of-the-art systems for the described challenge is that they are closed proprietary solutions. Sensor data from one system cannot be used with the system of another player on the market, as the data or the provided service are holding market value. There is a couple of manufacturers providing application programming interface for their sensors or trackers, however, most of them cannot be integrated into third-party software. The reason is the sensibility of personal or medical data, as their privacy cannot be guaranteed if they are offered for third parties via uncontrolled interfaces.

Combining the SensorHUB framework with medical sensors, we are concentrating on a method, which enables the collection of various kinds of health data from different sensor sources, and then utilizing the framework to infer the health status or find correlations and predictions.

A smartphone application is used as a gateway and controller for the measurements. Information about an ongoing measurement can be shown on the mobile device of the user, together with the final result and analysis at the end of the process. Users can utilize their own sensors or trackers for this process, but it is also possible to share sensors among many users. The data analysis and storage is done on a dedicated server. In order to insure scalability of the solution SensorHUB is used as the server-side backend system.

Special care has to be taken with regard to the security of personal data. The approach also requires a complex authentication system, which would encrypt medical data and authenticate the measurement device and measurement process at the same time.

We have designed and implemented the application on the top of SensorHUB, and have named it Sensible [Sensible]. We have selected a set of sensor types to be integrated into the system, both wired and wireless. Wireless sensors can harness the connectivity of the smartphone device of the users. In case of the wired sensors, there is an intermediary agent that receive the signals from those sensors, and load the data into the SensorHUB. We use Raspberry Pi devices for this task, running our software and the drivers of those sensors.

We believe that in the near future the sensors built on advanced technology will play an important role in efficient healthcare services and in early recognition of illnesses. Our results contribute to achieve this goal.

Besides the described domains, we are currently addressing two more domains, namely, the agriculture and the production line (Industry 4.0 or Industrial Internet). The architecture is similar: data is collected with domain-related sensors, locally processed and utilized, furthermore uploaded and analyzed. Services from their part are driven by the distilled data. These projects develop domain-specific solutions on top of the SensorHUB. Our experience shows that the aforementioned IoT projects, based on the utilized components and both the way and results of the development, validate the SensorHUB approach and its multi-domain capabilities. The reusability ratio of the framework components is rather high. The similarity in the architecture of the realized systems motivated us to apply a higher abstraction level development method, generate the configurable parts of the systems and increase product quality based on high-level validation methods. This led us to creation of a model-based development method.

6.3 Conclusions

I am sure that the worked out and presented methods and techniques contribute to the development of various effective solutions, which provide convenient, widely applicable, industrially relevant tools and methods for the verification and validation of model processors, as well as, effectively supports the application of the domain-specific modeling and model-processing techniques. All this is confirmed by the fact that most of the results have already been applied within various R&D projects and contribute to several strategic directions and activities at our department.

7 Summary

This chapter summarizes the main scientific results of the thesis, furthermore, provides the selected and closely related most important publications.

7.1 Thesis I: Methods for Verifying and Validating Graph Rewriting-Based Model Transformations

The results of Thesis I are the followings:

Classifying Model Transformation Approaches by Model Processing Properties

- *I have worked out the model transformation property classes, and I have applied them for supporting the classification of the verification and validation capabilities of model transformation approaches. Based on the property classes, I support to define the requirements against model transformation approaches and to categorize the already existing model transformation approaches and tools.*
- *The property classes (i) support the comparison of different verification and validation approaches and tools, (ii) support the identification of the appropriate verification/validation approaches and tools for a certain verification/validation challenge, and (iii) provide an overview about the research results and achievements of the graph transformation-based verification and validation.*

Method for Validating Rule-based Systems and Taming the Complexity of Model Transformation Verification/Validation Processes

- *Representing rule-based systems as graph rewriting systems, I have worked out a method for validating rule-based systems. I have shown that if a finite sequence of transformation rules with validating constraints (pre- and postconditions assigned to the rules) realizes a rule-based system and the execution of this sequence of transformation rules is successful for an input model, then the modified/output model satisfies the requirements defined by the validating constraints.*
- *I have designed a method for taming the complexity of model transformation verification/validation processes. I have shown that the taming method is applicable for rule-based systems represented by graph rewriting systems.*

Method for Test-Driven Verification/Validation

- *I have worked out both Basic and Advanced versions of the test-driven verification/validation method for model transformations defined with transformation rules and a control flow model. For the Basic version of the method, I have shown that generating valid instances of the input metamodel requires that the left-hand side (LHS) structures of the transformation rules be valid partial instances (result from Thesis III) of the input metamodel.*

Selected publications closely related to Thesis I: [1] [3] [5] [7] [10] [14] [17] [19] [21] [22] [23] [24] [25] [31] [36].

7.2 Thesis II: Model-Driven Methods Based on Domain-Specific Languages and Model Processors

The results of Thesis II are the followings:

Assuring the Quality of Software Development Projects by Applying Model-Driven Techniques and Model-Based Tools

- *I have worked out a method that supports the quality assurance of software development projects. The method utilizes model-driven techniques and model-based tools. I have worked out a method for relevant test scenario generation. I have shown that generated test scenarios cover the possible execution paths defined by the parameters and fixed variables, furthermore, these scenarios represent a restricted set of the whole area, therefore, the method results a more effective testing process. I have shown that, because of the domain-specific languages and automated model processing, the approach results a close relation between the software requirements and realized features.*

Method for Developing and Managing Domain-Specific Models and Method for Supporting the Transparent Switch between the Textual and Visual Views of Semantic Models

- *A method has been worked out that allows the definition and management of domain-specific models. The method provides suggestions regarding to the considerations and decisions we have to make during the analysis of the business needs and the definition of domain-specific languages, supports the steps and tasks related to the introduction of domain-specific languages and finally, helps during the maintenance of domain-specific languages.*
- *I have worked out an architecture that makes possible the effective and transparent switch between the visual and textual representations of semantic models.*

Method for Processing Mathworks Simulink Models with Graph Rewriting-Based Model Transformations

- *An integration method has been worked out between the Mathworks Simulink environment and the VMTS modeling and model processing framework. I have shown that the method for validating the domain-specific properties of software models (result from Thesis III) is an appropriate method for processing Simulink models with VMTS framework.*

Model-Driven Method for Managing Energy Efficient Operating Properties

- *A model-driven method has been worked out for managing the energy efficient operation properties of mobile devices. The method allows to define the energy efficiency properties on the level of software models. As a result of the solution, this aspect of software systems appears on the modeling level, which provides a more detailed view about the whole system. This supports both the more precise model-based analysis and the more relevant verification and validation of the software systems.*

Selected publications closely related to Thesis II: [4] [6] [8] [9] [16] [25] [29] [30] [32] [33] [34] [35] [38] [39].

7.3 Thesis III: Applying Domain-Specific Design Patterns and Validating Domain-Specific Properties

The results of Thesis III are the followings:

Method to Support Domain-Specific Design Patterns

- *I have worked out both the theoretical and practical basis to support the definition and management of domain-specific design patterns in metamodeling environments. I have shown that for metamodel level model transformation rules, a match for the left-hand side (LHS) of the transformation rule can be found with the instantiation relation, furthermore, a given part of the output model will be the instance of the right-hand side (RHS) of the transformation rule. It has been proven that a model allowing transitive containment with respect to its metamodel is not necessarily a partial instance. It has been shown that the relaxed instance models of a metamodel violates the instantiation rules in the following respects only: (i) relaxed multiplicity and cardinality, (ii) dangling edge relaxation, and (iii) incomplete attributes.*

Method for Validating the Domain-Specific Properties of Software Models

- *I have worked out a method for validating domain-specific properties of software models during model processing. I have shown that success conditions and negative success conditions have no effect on the result of the model processing. I have proven that validating transformation rules do not modify the output model.*

Algorithms for Handling the Validating Constraints in a Modular Way

- *I have worked out a method for consistent handling of validating constraints. I have shown that in case of model processing the crosscutting nature of validating constraints cannot be always eliminated. I have worked out algorithms which semi-automatically identify the crosscutting constraints in model transformations (coloring algorithm), furthermore, which, based on the coloring, extracts the repetitive and crosscutting constraints and generates the constraint call definitions.*

Selected publications closely related to Thesis III: [2] [8] [10] [11] [12] [13] [15] [18] [20] [26] [27] [28] [37] [40].

In summary, I highlight that the provided methods, algorithms and solutions are about to utilize domain-specific modeling and model-driven techniques to contribute in achieving more efficient and errorless software development processes. The goal was to provide methods and tools to support those parts of the development cycle, which can be atomized, and to make more quality software artifacts. I believe that the results introduced in the thesis are contributing to support effective validation of model processors and the domain-specific properties of various software models.

Publications Closely Related to the Thesis

[1] **L. Lengyel**, T. Levendovszky, H. Charaf, Validated model transformation-driven software development, International Journal of Computer Applications in Technology, pp. 106-119, 2008.

[2] T. Levendovszky, **L. Lengyel**, T. Mészáros, Supporting domain-specific model patterns with metamodeling, Software and System Modeling 8:(4), pp. 501-520, 2009. **IF: 1,533**

[3] M. Asztalos, I. Madari, **L. Lengyel**, Towards Formal Analysis of Multi-paradigm Model Transformations, Simulation-Transactions of the Society for Computer simulation International 86:(7), pp. 429-452, 2010. **IF: 0,611**

[4] **L. Lengyel**, G. Mezei, Model-Driven Paradigms – The Evolution of a University Course., 8th edition of the Educators' Symposium, Innsbruck, Austria, ACM, pp. 13-20. 2012.

[5] M. Asztalos, **L. Lengyel**, T. Levendovszky, Formal specification and analysis of functional properties of graph rewriting-based model transformation, Software Testing Verification & Reliability 23:(5), pp. 405-435, 2013. **IF: 1,2**

[6] H. Charaf, P. Ekler, T. Mészáros, I. Kelényi, B. Kővári, I. Albert, B. Forstner, **L. Lengyel**, Mobile Platforms and Multi-Mobile Platform Development, Acta Cybernetica 21:(4), pp. 529-552, 2014.

[7] **L. Lengyel**, H. Charaf, Test-driven verification/validation of model transformations, Frontiers of Information Technology & Electronic Engineering 16:(2), pp. 85-97, 2015.

[8] **L. Lengyel**, T. Meszaros, M. Asztalos, P. Boros, A. Mate, G. Madacs, P. Hudak, K. Kovacs, A. Tresch, H. Charaf, Quality Assured Model-Driven Requirements Engineering and Software Development, The Computer Journal 58:(11), pp. 3171-3186, 2015. **IF: 1,0**

[9] **L. Lengyel**, P. Ekler, T. Ujj, T. Balogh, H. Charaf, SensorHUB – An IoT Driver Framework for Supporting Sensor Networks and Data Analysis, International Journal of Distributed Sensor Networks 2015, Article ID 454379, 12 pages, 2015. **IF: 0,906**

[10] **L. Lengyel**, H. Charaf, Open Issues in Model Transformations for Multimodal Applications, Journal of Multimodal User Interfaces 9:(4), pp. 377-385, 2015. **IF: 1,017**

[11] **L. Lengyel**, T. Levendovszky, T. Mészáros, H. Charaf, Supporting design patterns in graph rewriting-based model transformation, 2nd Int. Working Conference on Evaluation of Novel Approaches to software Engineering, Barcelona, pp. 25-32, 2007.

[12] **L. Lengyel**, T. Levendovszky, H. Charaf, Applying multi-paradigm modeling to multi-platform mobile development, Workshop on Multi-Paradigm Modeling: Concepts and Tools, Nashville, USA, pp. 9-21, 2007.

[13] **L. Lengyel**, T. Levendovszky, H. Charaf, Identification of crosscutting concerns in constraint-driven validated model transformations, Third Workshop on Models and Aspects - Handling Crosscutting Concerns, Berlin, Germany, pp. 15-20, 2007.

[14] **L. Lengyel**, T. Levendovszky, G. Mezei, T. Vajk, H. Charaf, Practical uses of validated model transformation, Eurocon 2007 - The International Conference on Computer as a Tool, Warsaw, Poland, pp. 2200-2207, 2007.

[15] T. Levendovszky, **L. Lengyel**, G. Mezei, T. Mészáros, Introducing the VMTS mobile toolkit., 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007. Kassel, Germany, pp. 587-592, 2008.

[16] L. Angyal, M. Asztalos, **L. Lengyel**, T. Levendovszky, I. Madari, G. Mezei, T. Mészáros, L. Siroki, T. Vajk, Towards a fast, efficient and customizable domain-specific modeling framework, IASTED International Conference on Software Engineering, Innsbruck, Austria, pp. 11-16, 2009.

[17] M. Asztalos, **L. Lengyel**, T. Levendovszky, A formalism for describing modeling transformations for verification, 6th International Workshop on Model-Driven Engineering, Verification and Validation: MoDeVVa '09. Denver, USA, pp. 1-10, 2009.

[18] M. Asztalos, T. Mészáros, **L. Lengyel**, Generating executable BPEL code from BPMN models, 5th International Workshop on Graph-Based Tools - Graph Transformation Tool Contest. Zurich, Swiss, 2009.

[19] M. Asztalos, **L. Lengyel**, T. Levendovszky, Toward Automated Verification of Model Transformations: A Case Study of Analysis of Refactoring Business Process Models, Electronic Communications of the EASST 21, pp. 1-5, 2009.

[20] **L. Lengyel**, T. Levendovszky, L. Angyal, Identification of Crosscutting Constraints in Metamodel-Based Model Transformations, International IEEE Conference Devoted to the 150-Anniversary of Alexander S Popov: Eurocon 2009. Saint-Petersburg, Italy, pp. 359-364, 2009.

[21] M. Asztalos, P. Ekler, **L. Lengyel**, T. Levendovszky, T. Mészáros, G. Mezei, Automated Verification by Declarative Description of Graph Rewriting-Based Model Transformations, Electronic Communications of the EASST 42:(12), 2010.

[22] M. Asztalos, **L. Lengyel**, T. Levendovszky, Towards Automated, Formal Verification of Model Transformations, 3rd International Conference on Software Testing, Verification and Validation, Paris, France, pp. 15-24, 2010.

[23] M. Asztalos, P. Ekler, **L. Lengyel**, T. Levendovszky, Verification of Model Transformations to Refactoring Mobile Social Networks, Electronic Communications of the EASST 32:(12), 2010.

[24] I. Madari, M. Asztalos, T. Mészáros, **L. Lengyel**, H. Charaf, Implementing QVT in a domain-specific modeling framework, 5th International Conference on Software and Data Technologies, Athens, Greece, pp. 304-307, 2010.

[25] P. Fehér, T. Mészáros, P. Mosterman, **L. Lengyel**, Processing Simulink Models with Graph Rewriting-Based Model Transformation, ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012: Tutorials, Innsbruck, Austria, 2012.

[26] **L. Lengyel**, H. Charaf, Validating domain-specific properties with graph transformations, International Conference on Innovative Technologies, Rijeka, Croatia, pp. 145-148, 2012.

[27] **L. Lengyel**, The Role of Graph Transformations in Validating Domain-Specific Properties, International Journal of Computer Engineering and Technology 3:(3), pp. 406-425, 2012.

[28] **L. Lengyel**, Modularized Constraint Management in Model Transformation Frameworks, Acta Polytechnica Hungarica 10:(1), pp. 101-119, 2013. **IF: 0,471**

[29] P. Fehér, T. Mészáros, P.J. Mosterman, **L. Lengyel**, A Novel Algorithm for Flattening Virtual Subsystems in Simulink Models, Proceedings of the IEEE International Conference on System Science and Engineering, Budapest, Hungary, pp. 369-375, 2013.

[30] P. Fehér, T. Mészáros, Pieter J Mosterman, **L. Lengyel**, Flattening Virtual Simulink Subsystems with Graph Transformation, Workshop on Complex Systems Modelling and Simulation, Milan, Italy, pp. 39-60, 2013.

[31] T. Mészáros, P. Fehér, **L. Lengyel**, Visual Debugging Support for Graph Rewriting-based Model Transformations, International Conference on Computer as a Tool, Eurocon 2013, Zagreb, Croatia, pp. 482-487, 2013.

[32] G. Kövesdán, M. Asztalos, **L. Lengyel**, A Classification of Domain-Specific Language Intents, International Journal of Modeling and Optimization 4:(1) pp. 67-73, 2014.

[33] G. Kövesdán, M. Asztalos, **L. Lengyel**, Architectural Design Patterns for Language Parsers, Acta Polytechnica Hungarica 11:(5) pp. 39-57, 2014. **IF: 0,649**

[34] G. Mezei, **L. Lengyel,** T. Mészáros, T. Vajk, Metamodeling, Model Processing and Simulation – Putting the Puzzle Pieces Together Based on Industrial Experiences, ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, MODELS 2014: Tutorials, Valencia, Spain, 2014.

[35] I. Kelényi, J. K. Nurminen, M. Siekkinek, **L. Lengyel**, Supporting Energy-Efficient Mobile Application Development with Model-Driven Code Generation, Advanced Computational Methods for Knowledge Engineering: Proceedings of the 2nd International Conference on Computer Science, Applied Mathematics and Applications, Budapest, Hungary, pp. 143-156, 2014.

[36] **L. Lengyel**, Validating Rule-based Algorithms, Acta Polytechnica Hungarica 12:(4), pp. 59-75, 2015. **IF: 0,544**

[37] G. Kövesdán, M. Asztalos, **L. Lengyel**, Aggregate Callback: A Design Pattern for Flexible and Robust Runtime Model Building, 3rd International Conference on Model-Driven Engineering and Software Development, Angers, France, pp. 149-156, 2015.

[38] P. Fehér, M. Asztalos, T. Mészáros, **L. Lengyel**, A MapReduce-based Approach for Finding Inexact Patterns in Large Graphs, MODELSWARD 2015: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, Angers, France, pp. 205-212, 2015.

[39] P. Fehér, M. Asztalos, T. Vajk, T. Mészáros, **L. Lengyel**, Detecting subgraph isomorphism with MapReduce, Journal of Supercomputing 73:(5), pp. 1810-1851, 2017. **IF: 1,088**

[40] **L. Lengyel**, P. Ekler, I. Tömösvári, T. Balogh, G. Mezei, B. Forstner, H. Charaf, Model-Driven Multi-Domain IoT, Book chapter in Emerging Trends and Applications of the Internet of Things, IGI Global, pp. 167-191, 2017.

[41] H. Hejazi, H. Rajab, T. Cinkler, **L. Lengyel**, Survey of Platforms for Massive IoT, 2018 IEEE International Conference on Future IoT Technologies, Hungary, Paper 8, 2018.

[42] G. Kövesdán, **L. Lengyel**, Meta3: A Code Generator Framework for Domain-Specific Languages, Software and Systems Modeling, Accepted, 2018. **IF: 1,654**

Bibliography

[Abramski et al, 1993] S. Abramski, M.D. Gabbay and T.S.E. Maibaum (ed), Handbook of Logic in Computer Science 2, Oxford University Press, 1993.

[AGG] AGG: The Attributed Graph Grammar System, http://tfs.cs.tu-berlin.de/agg

[Agile Manifesto, 2001] Manifesto for Agile Software Development, 2001, http://agilemanifesto.org/

[Akehurst and Kent, 2002] D. Akehurst and S. Kent, A Relational Approach to Defining Transformations in a Metamodel, In UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, LNCS 2460, Springer-Verlag, pp. 243-258, 2002.

[Amrani et al, 2012] M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani and M. Wimmer, Towards a model transformation intent catalog, In Proceedings of the First Workshop on the Analysis of Model Transformations, ACM, New York, NY, USA, pp. 3-8, 2012.

[Anand et al, 2004] M. Anand, E. B. Nightingale and J. Flinn, Ghosts in the machine: interfaces for better power management, in Proceedings of the 2nd international conference on Mobile systems, applications, and services, MobiSys '04, ACM, pp. 23-35, 2004.

[Anand et al, 2005] M. Anand, E. B. Nightingale and J. Flinn, Self-tuning wireless network power management, Wirel. Netw 11, pp. 451-469, 2005.

[Anastasakis et al, 2007] K. Anastasakis, B. Bordbar, G. Georg and I. Ray, UML2Alloy: a challenging model transformation, Proceedings of the MoDELS07, LNCS 4735, Springer, pp. 436-450, 2007.

[Apache Cassandra] Apache Cassandra, database, 2016, http://cassandra.apache.org

[Apache Flume] Apache Flume, service for collecting, aggregating, and moving large amounts of log data, 2017, https://flume.apache.org

[Apache Hadoop] The Apache Software Foundation, Apache Hadoop, 2017, http://hadoop.apache.org/

[Apache Hive] Apache Hive, data warehouse platform, 2017, https://hive.apache.org

[Apache Kafka] Apache Kafka, publish-subscribe messaging rethought, 2017, http://kafka.apache.org

[Apache Spark] Apache Spark, General engine for large-scale data processing, 2017, https://spark.apache.org

[Assmann, 1996] U. Assmann, How to Uniformly specify Program Analysis and Transformation with Graph Rewrite Systems, Proceedings of the 6 Int. Conference on Compiler Construction, LNCS 1060, Springer, 1996.

[Assmann, 2000] U. Assmann, Graph rewrite systems for program optimization, ACM TOPLAS 22, pp. 583-637, 2000.

[Assmann and Ludwig, 2000] U. Assmann and A. Ludwig, Aspect Weaving by Graph Rewriting, Generative Componentbased Software Engineering, Lecture Notes in Computer Science 1799, Springer, 2000.

[ATL] ATL: ATLAS Transformation Language, 2018, http://eclipse.org/atl/

[AToM3] AToM^3: A Tool for Multi-paradigm, Multi-formalism and Meta-modeling, http://atom3.cs.mcgill.ca

[AWS IoT] AWS IoT, 2017, https://aws.amazon.com/iot/

[Azure IoT Suite] Azure IoT Suite, 2018, https://azure.microsoft.com/en-us/suites/iot-suite/

[Barbosa et al, 2009] P. Barbosa, F. Ramalho, J. Figueiredo, A. Junior, A. Costa and L. Gomes, Checking semantics equivalence of MDA transformations in concurrent systems, Journal of Universal Computer Science 15(11), pp. 2196–2224, 2009.

[Baresi et al, 2003] L. Baresi, R. Heckel, S. Thöne and D. Varró, Modeling and Analysis of Architectural Styles, Proc ESEC 2003: 9th European Software Engineering Conference, Helsinki, Finland, ACM Press, pp. 68-77, 2003.

[Barr and Asanovic, 2006] K. C. Barr and K. Asanovic, Energy-aware lossless data compression, ACM Transactions on Computer Systems 24:(3), pp. 250-291, 2006.

[BBC Research, 2017] BBC Research, Global markets and technologies for sensors, BBC Research Report, March 2017, https://www.bccresearch.com/market-research/instrumentation-and-sensors/sensors-technologies-markets-report-ias006h.html

[Beuche et al, 2006] D. Beuche and M. Dalgarno, Software product line engineering with feature models, Methods and Tools, 2006.

[Biermann et al, 2011] E. Biermann, C. Ermel and G. Taentzer, Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation, Software and Systems Modeling 11:(2), Springer, pp. 227-250, 2011.

[Bisztray et al, 2008a] D. Bisztray, R. Heckel and H. Ehrig, Verification of architectural refactorings by rule extraction, In Fundamental Approaches to Software Engineering, LNCS 4961, Springer, pp. 347-361, 2008.

[Bisztray et al, 2008b] D. Bisztray, K. Ehrig, R. Heckel, P. Torrini, A. Corradini, B. König, P. Baldan and L. Baresi, Formal Analysis of Model Transformations, Software Engineering for Service-Oriented Overlay Computers, Sensoria 016004 35, pp. 178-193, 2008.

[Blostein et al, 1996] D. Blostein, H. Fahmy and A. Grbavec, Issues in the practical use of graph rewriting. In proceedings of the 5th International Workshop on Graph Grammars and Their Application to Computer Science, Williamsburg, USA, LNCS 1073, Springer-Verlag, pp. 38-55, 1996.

[Bottoni et al, 2000] P. Bottoni, G. Taentzer and A. Schürr, Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation, Proceedings of the Visual Languages 2000 IEEE Computer Society, pp. 59-60, 2000.

[Braun and Marschall, 2003] P. Braun and F. Marschall, BOTL - The bidirectional object-oriented transformation language, Fakultat fur Informatik, Technische Universitat Munchen, Technical Report TUMI0307, 2003.

[Bundy, 1986] A. Bundy, The computer modelling of mathematical reasoning, Academic Press, 1986.

[Cabot et al, 2008] J. Cabot, R. Clariso and D. Riera, Verification of UML/OCL class diagrams using constraint programming, In MoDeVVa 2008, ICST Workshop, pp. 73-80, 2008.

[Cabot et al, 2010] J. Cabot, R. Clariso, E. Guerra and J. de Lara, Verification and Validation of Declarative Model-to-Model Transformations Through Invariants, Journal of Systems and Software 83:(2), pp. 283-302, 2009.

[Clifton et al, 2014] L. Clifton, D.A. Clifton, M.A.F. Pimentel, P.J. Watkinson and L. Tarassenko, Predictive Monitoring of Mobile Patients by Combining Clinical Observations with Data from Wearable Sensors, Biomedical and Health Informatics, IEEE Journal 18:(3), pp. 722-730, 2014.

[Czarnecki and Eisenecker, 2000] K. Czarnecki and U.W. Eisenecker, Generative programming: methods, tools, and applications, Addison-Wesley, 2000.

[Czarnecki and Helsen, 2006] K. Czarnecki and S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45:(3), pp. 621-646, 2006.

[Davies, 2011] R. Davies, Non-Functional Requirements: Do User Stories Really Help?, http://www.methodsandtools.com/archive/archive.php?id=113, 2011.

[Dijkstra, 1976] E.W. Dijkstra, A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.

[Docker] Docker, A platform for distributed applications, 2018, https://www.docker.com

[Dong and Zhong, 2011] M. Dong and L. Zhong, Chameleon: A color-adaptive web browser for mobile oled displays, In Proceedings of the 9th international conference on Mobile systems, applications, and services, MobiSys '11, ACM 11:(5), pp. 85-98, 2011.

[Donohoe, 2012] P. Donohoe (ed.), Software Product Lines: Experience and Research Directions, Springer Science & Business Media, 532 pages, 2012.

[Eclipse] Eclipse Framework, 2018, http://www.eclipse.org/

[Eclipse GMP] Eclipse Graphical Modeling Project, 2018, http://www.eclipse.org/modeling/gmp/

[Eclipse EEF] Eclipse Extended Editing Framework, 2018, http://www.eclipse.org/modeling/emft/?project=eef

[Ehrig et al, 1991a] H. Ehrig, A. Habel, H.-J. Kreowski and F. Parisi-Presicce, From Graph Grammars to High Level Replacement Systems, In Graph Grammars and Their Application to Computer Science, LNCS 532, Springer, pp. 269-291, 1991.

[Ehrig et al, 1991b] H. Ehrig, A. Habel, H.-J. Kreowski and F. Parisi-Presicce, Parallelism and Concurrency in High-Level Replacement Systems, Mathematical Structures in Computer Science 1:(3), pp. 361-404, 1991.

[Ehrig et al, 1999] H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozemberg (ed.), Handbook on graph grammars and computing by graph transformation: Application, languages and tools, World Scientific 2, Singapore, 1999.

[Ehrig et al, 2004] H. Ehrig, A. Habel, J. Padberg and U. Prange, Adhesive High-Level Replacement Categories and Systems, In proceedings of the ICGT 2004, LNCS 3256, Springer, pp. 144-160, 2004.

[Ehrig et al, 2005] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró and Sz. Varró-Gyapay, Termination Criteria for Model Transformation, FASE 2005, LNCS, pp. 49-63, 2005.

[Ehrig et al, 2006] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer, Fundamentals of algebraic graph transformation, Monographs in Theoretical Computer Science, Springer, 2006.

[Ehrig et al, 2010] H. Ehrig, A Habel, L. Lambers, F. Orejas and U. Golas, Local Confluence for Rules with Nested Application Conditions, Proceedings of ICGT'10, Springer, LNCS 6372, pp. 330-345, 2010.

[EIT Climate-KIC] EIT Climate-KIC, Knowledge & Innovation Community, http://www.climate-kic.org/

[Ekler et al, 2015] P. Ekler, T. Balogh, T. Ujj, H. Charaf and L. Lengyel, Social Driving in Connected Car Environment, European Wireless 2015, 21th European Wireless Conference, Budapest, Hungary, pp. 136-141, 2015.

[Flores and Fillottrani, 2003] A. Flores and P. Fillottrani, Evaluation framework for Design pattern formal models, Proceedings of the CACIC'03 IX Argentinean Conference on Computer Science, La Plata, Argentina, 2003.

[Fowler, 2010] M. Fowler, Domain-specific languages, Addison-Wesley Professional, 2010.

[Fujaba] Fujaba Tool Suite, 2012, http://www.fujaba.de/

[Gamma et al, 1995] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: Elements of reusable object-oriented software, Addison-Wesley Professional Computing Series, 1995.

[Giese et al, 2006] H. Giese, S. Glesner, J. Leitner, W. Schafer and R. Wagner, Towards verified model transformations, In ModeVVa06, 2006.

[v. Gorp et al, 2003] P. v. Gorp, H. Stenten, T. Mens and S. Demeyer, Towards Automating Source-Consistent UML Refactorings, In UML 2003 - The Unified Modeling Language. Modeling Languages and Applications, 6th International Conference, San Francisco, USA, LNCS 2863, Springer- Verlag, pp. 144-158, 2003.

[GReAT] GReAT: Graph Rewriting and Transformation, http://www.isis.vanderbilt.edu/tools/GReAT

[GROOVE] GROOVE: GRaphs for Object-Oriented VErification, http://groove.sourceforge.net/groove-index.html

[Guerra and de Lara, 2007] E. Guerra and J. de Lara, Event-driven grammars: Relating abstract and concrete levels of visual languages, Software and System Modeling 6:(3), pp. 317-347, 2007.

[Habel et al, 1996] A. Habel, R. Heckel and G. Taentzer, Graph grammars with negative application conditions, Fundamenta Informaticae 26, pp. 287-313, 1996.

[Hausmann et al, 2002] J.H. Hausmann, R. Heckel and S. Sauer, Extended Model Relations with Graphical Consistency Conditions, In UML 2002 Workshop on Consistency Problems in UML-based Software Development, Blekinge Institute of Technology 6174, 2002.

[Hayes-Roth, 1985] F. Hayes-Roth, Rule-based systems, Communication of ACM 28:(9), pp. 921-932, 1985.

[Heckel et al, 2002] R. Heckel, J.M. Küster and G. Taentzer, Confluence of Typed Attributed Graph Transformation Systems, Proceedings of the 1st international conference of graph transformation, LNCS 2505, Springer, Berlin Heidelberg New York, pp. 161-176, 2002.

[Hetzel, 1988] W.C. Hetzel, The Complete Guide to Software Testing, 2nd ed., Wellesley, Mass. QED Information Sciences, ISBN: 0894352423, 280 pages, 1988.

[Hoque et al, 2013] M. Hoque, M. Siekkinen and J. K. Nurminen, TCP receive buffer aware wireless multimedia streaming - an energy efficient approach, in Proceedings of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV'13, ACM, pp. 13-18, 2013.

[Huet, 1980] G. Huet, Confluent reductions, Abstract properties and applications to term rewriting systems, Journal of the ACM 27:(4), pp. 797-821, 1980.

[Hulsbusch et al, 2010] M. Hulsbusch, B. Konig, A. Rensink, M. Semenyak, C. Soltenborn and H. Wehrheim, Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques, Integrated Formal Methods, Springer, LNCS 6396, pp. 183-198, 2010.

[Humbly, 2006] C. Humbly, Data is the New Oil, ANA Senior marketer's summit, Kellogg School, 2006.

[IEEE Standard Glossary, 1990] IEEE Standard Glossary of Software Engineering Terminology, 610.12-1990, 1990.

[Kaner, 2006] C. Kaner, Exploratory Testing, Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, 2006.

[Kaner et al, 1990] C. Kaner, J. Falk and H.Q. Nguyen, Testing Computer Software, 2nd ed. New York, John Wiley and Sons, Inc., ISBN 0-471-35846-0, 480 pages, 1990.

[Karsai et al, 2003] G. Karsai, A. Agrawal and F. Shi, On the Use of Graph Transformation in the Formal Specification of Model Interpreters, Journal of Universal Computer Science 9(11), Special issue on Formal Specification of CBS, pp. 1296-1321, 2003.

[Kelly and Tolvanen, 2008] S. Kelly and J.P. Tolvanen, Domain-specific modeling: enabling full code generation, Wiley-IEEE Computer Society Pr, 2008.

[Kolawa and Huizinga, 2007] A. Kolawa and D. Huizinga, Automated Defect Prevention, Best Practices in Software Management. Wiley-IEEE Computer Society Press, ISBN 0-470-04212-5, pp. 41-43, 2007.

[König and Kozioura, 2008] B. König and V. Kozioura, Augur 2 - A New Version of a Tool for the Analysis of Graph Transformation Systems, Electronic Notes in Computer Science 211, pp. 201-201, 2008.

[Küster, 2006] J.M. Küster, Definition and validation of model transformations, Software and Systems Modeling 5:(3), pp. 233-259, 2006.

[Lack and Sobocinski, 2004] S. Lack and P. Sobocinski, Adhesive Categories, Proceedings of FOSSACS 2004, LNCS 2987, Springer, pp. 273-288, 2004.

[de Lara and Taentzer, 2004] J. de Lara and G. Taentzer, Automated Model Transformation and its Validation with AToM3 and AGG, in Diagrammatic Representation and Inference, Lecture Notes in Artificial Intelligence 2980, Springer, pp. 182-198, 2004.

[de Lara et al, 2004] J. de Lara, H. Vangheluwe and M. Alfonseca, Metamodelling and graph grammars for multi-paradigm modelling in AToM^3, Journal of Software and Systems Modeling 3:(3), pp. 194-209, 2004.

[de Lara and Guerra, 2009] J. de Lara and E. Guerra, Formal Support for QVT-Relations with Coloured Petri Nets, In Proceedings of the MODELS'09, LNCS 5795, Denver, USA, pp. 256-270, 2009.

[Leitner et al, 2007] A. Leitner, I. Ciupa, M. Oriol, B. Meyer and A. Fiva, Contract Driven Development = Test Driven Development – Writing Test Cases, Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Croatia, 2007.

[Lengyel, 2006] L. Lengyel, Online Validation of Visual Model Transformations, PhD thesis, Budapest University of Technology and Economics, Department of Automation and Applied Informatics, 2006.

[Lengyel et al, 2015] L. Lengyel, P. Ekler, T. Ujj, T. Balogh, H. Charaf, Zs. Szalay and L. Jereb, ICT IN ROAD VEHICLES – The VehicleICT Platform, 4th International Conference on Models and Technologies for Intelligent Transportation Systems, Budapest, Hungary, pp. 457-462, 2015.

[Levendovszky et al, 2007] T. Levendovszky, U. Prange and H. Ehrig, Termination Criteria for DPO Transformations with Injective Matches, Electronic Notes in Theoretical Computer Science 175:(4), pp. 87-100. 2007.

[Massoni et al, 2006] T. Massoni, R. Gheyi and P. Borba, An approach to invariant-based program refactoring, In Software Evolution through Transformations 2006. Electronic Communications of the EASST, 2006.

[Mens et al, 2002] T. Mens, S. Demeyer and D. Janssens, Formalising behaviour preserving program transformations, Proceedings of the First International Conference on Graph Transformation, London, UK, Springer-Verlag, pp. 286-301, 2002.

[Mens and Tourwe, 2004] T. Mens and T. Tourwe, A Survey of Software Refactoring, IEEE Transactions on Software Engineering 30:(2), pp. 126-139, 2004.

[Mens and v. Gorp, 2006] T. Mens and P. v. Gorp, A taxonomy of model transformation, Electron. Notes Theoretical Computer Science 152, pp. 125-142, 2006.

[Microsoft T4] Microsoft T4, Code Generation and T4 Text Templates, https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates

[Mosterman et al, 2004] P. J. Mosterman, J. Sztipanovits and S. Engell, Computer-automated multiparadigm modeling in control systems technology, IEEE Transactions on Control Systems Technology 12:(2), pp. 223–234, march 2004.

[Mosterman and Vangheluwe, 2000] P. J. Mosterman and H. Vangheluwe, Computer automated multi-paradigm modeling in control system design. IEEE Transactions on Control System Technology 12, pp. 65–70, 2000.

[MQTT] MQTT, A machine-to-machine (M2M)/"Internet of Things" connectivity protocol, http://mqtt.org

[Narayanan and Karsai, 2008] A. Narayanan and G. Karsai, Towards verifying model transformations, ENTCS 211, pp. 191-200, 2008.

[Newman, 1942] M.H.A. Newman, On theories with a combinatorial definition of equivalence, In Annals of Mathematics 43:(2), pp. 223-243, 1942.

[Node.js] Node.js, https://nodejs.org/

[Norbisrath, 2013] U. Norbisrath, R. Jubeh and A. Zündorf, Story Driven Modeling, CreateSpace Independent Publishing Platform, 348 pages, 2013.

[Nurminen, 2010] J. Nurminen, Parallel connections and their effect on the battery consumption of a mobile phone, in Consumer Communications and Networking Conference, 2010 7th IEEE, pp. 1-5, 2010.

[Nurminen and Noyranen, 2009] J. Nurminen and J. Noyranen, Parallel data transfer with voice calls for energy-efficient mobile services, in Mobile Wireless Middleware, Operating Systems, and Applications, Lecture Notes of the Institute for Computer Sciences 7, Social Informatics and Telecommunications Engineering, Springer Berlin Heidelberg, pp. 87-100, 2009.

[OMG MDA, 2014] OMG Model-Driven Architecture (MDA), 2014, http://www.omg.org/mda

[OMG OCL, 2014] OMG Object Constraint Language Specification, Object Management Group OMG document formal/2014-02-03, 2014, http://www.omg.org/spec/OCL/

[OMG QVT, 2016] OMG Query/View/Transformation (QVT) Specification, Meta Object Facility 2.0 Query/Views/Transformation Specification, OMG document formal/2016-06-03, 2016, http://www.omg.org/spec/QVT/

[OMG UML, 2015] OMG UML specification, version 2.5, OMG document formal/15-03-01, 2015, http://www.omg.org/spec/UML/

[Pan, 1999] J. Pan, Software Testing, 18-849b Dependable Embedded Systems, Carnegie Mellon University, 1999.

[Pantelopoulos and Bourbakis, 2010] A. Pantelopoulos and N.G. Bourbakis, A survey on wearable sensor-based systems for health monitoring and prognosis, IEEE Trans. Systems, Man, and Cybernetics, Part C: Applications and Reviews 40:(1), pp. 1-12, 2010.

[Plump, 1998] D. Plump, Termination of graph rewriting is undecidable, In Fundam. Inf., Vol. 33(2), Amsterdam, The Netherlands: IOS Press, pp. 201-209, 1998.

[Plump, 2005] D. Plump, Confluence of graph transformation revisited, LNCS 3838, Springer, pp. 280-308, 2005.

[Pohl, 2010] K. Pohl, Requirements Engineering: Fundamentals, Principles, and Techniques, Springer, 2010.

[P/Invoke] Marshaling Data with Platform Invoke, https://docs.microsoft.com/en-us/dotnet/framework/interop/marshaling-data-with-platform-invoke

[Qualcomm, 2012] Qualcomm Inc., Managing background data traffic in mobile devices, 2012, http://www.qualcomm.com/media/documents/managing-background-data-traffic-mobile-devices

[Rensink et al, 2004] A. Rensink, A. Schmidt and D. Varró, Model Checking Graph Transformations: A Comparison of Two Approaches. Proceedings of the ICGT 2004: Second International Conference on Graph Transformation, LNCS 3256, Springer, Rome, Italy, pp. 226-241, 2004.

[Rozenberg, 1997] G. Rozenberg (ed.), Handbook on graph grammars and computing by graph transformation: Foundations 1, World Scientific, Singapore, 1997.

[Runge et al, 2011] O. Runge, C. Ermel and G. Taentzer, AGG 2.0 - New Features for Specifying and Analyzing Algebraic Graph Transformations, AGTIVE 2011, International Symposium on Applications of Graph Transformation with Industrial Relevance, Budapest, Hungary, 2011.

[Samek, 2002] M. Samek, Practical statecharts in C/C++, CMP Books, ISBN 1578201101, 2002.

[Schatz, 2008] B. Schatz, Formalization and Rule-Based Transformation of EMF Ecore-Based Models, Software Language Engineering: First International Conference, SLE 2008, France, pp. 227-244, 2008.

[Schürr, 1994] A. Schürr, Specification of graph translators with triple graph grammars, Proceedings of the WG94 international workshop on graph-theoretic concepts in computer science, LNCS 903, Springer, Berlin Heidelberg New York, pp. 151-163, 1994.

[Sendall and Kozaczynski, 2003] S. Sendall and W. Kozaczynski, Model transformation: the heart and soul of model-driven software development, IEEE Software 20, pp. 42-45, 2003.

[Sensible] Sensible Project - IoT in Healthcare, https://www.aut.bme.hu/Pages/Research/Sensible

[Sensing IoT, 2015] Sensing the future of the Internet of Things, 2015, http://www.pwc.com/us/en/increasing-it-effectiveness/assets/future-of-the-internet-of-things.pdf

[SensorHUB] The SensorHUB project, https://www.aut.bme.hu/SensorHUB/

[Simulink] Simulink. https://www.mathworks.com/products/simulink.html

[Slash Data, 2017a] Slash Data, IoT Developer Landscape – The backgrounds, endeavours, challenges, and results of IoT developers, 2017, https://www.slashdata.co/reports/iot-developer-landscape

[Slash Data, 2017b] Slash Data, How and where to reach IoT developers – Which communities, events, and other channels do IoT developers use to stay up to date?, 2017, https://www.slashdata.co/reports/how-and-where-to-reach-iot-developers

[SOLSUN] The SOLSUN project, http://solsun.co.uk/index.php/SOLSUN/

[Sommerville and Kotonya, 1998] I. Sommerville and G. Kotonya, Requirements Engineering: Processes and Techniques, John Wiley & Sons, Inc. New York, NY, USA, 1998.

[Sutton and Rouvellou, 2002] S.M. Sutton, and I. Rouvellou, Modeling of Software Concerns in Cosmos, In Proceedings of the 1st International Conference on Aspect-Oriented Software Development, ACM Press, pp. 127-133, 2002.

[Swagger] Swagger, RESTful API representation, 2017, http://swagger.io

[Swan 2013] M. Swan, The Quantified Self: Fundamental Disruption in Big Data Science and Biological Discovery, Big Data 1:(2), pp. 85-99, 2013.

[Syriani, 2009] E. Syriani, Matters of model transformation, McGill University, no. SOCS-TR-2009.2, School of Computer Science, 2009.

[Sztipanovits et al, 1997] J. Sztipanovits and G. Karsai, Model-integrated computing, IEEE Computer, pp. 110-112, 1997.

[Taentzer et al, 2005] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varró and Sz. Varró-Gyapay, Model Transformation by Graph Transformation: A Comparative Study, Proceedings of the International Workshop on Model Transformations in Practice, MTiP 2005, Montego Bay, Jamaica, pp. 1-48, 2005.

[Thibodeau, 2014] P. Thibodeau, The ABCs of the Internet of Things, Computerworld US, 2014, http://www.techworld.com/networking/abcs-of-internet-of-things-3516134/3/

[URBMOBI] The URBMOBI project, http://www.climate-kic.org/case-studies/urban-resistance-to-the-effects-of-climate-change/

[Vajk et al, 2009] T. Vajk, R. Kereskényi, T. Levendovszky and Á. Lédeczi, Raising the abstraction of domain-specific model translator development, 16th IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems, San Francisco, USA, pp. 31-37, 2009.

[Varró, 2004] D. Varró, Automated formal verification of visual modeling languages by model checking, Journal on Software and System Modeling 3(2), pp. 85-113, 2004.

[Varró and Pataricza, 2003] D. Varró and A. Pataricza, Automated formal verification of model transformations, Proceedings of theUML03 workshop, Technical Report, pp. 63-78, 2003.

[Varró et al, 2003] D. Varró and A. Pataricza, VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML, Journal of Software and Systems Modeling, 2003.

[Varró et al, 2006] D. Varró, Sz. Varró-Gyapay, H. Ehrig, U. Prange and G. Taentzer, Termination Analysis of Model Transformations by Petri Nets, Lecture Notes in Computer Science 4178, pp. 260-274, 2006.

[VehicleICT] The VehicleICT project, https://www.aut.bme.hu/VehicleICT/

[VIATRA2] VIATRA2 (VIsual Automated model TRAnsformations) framework, http://eclipse.org/gmt/VIATRA2

[Vision Mobile, 2015a] Vision Mobile, IoT Report series: The Wearables Landscape 2015 - Developer and Platform Leaderboard for Wearables, 2015, http://www.visionmobile.com/product/iot-report-series-wearables-landscape-2015/

[Vision Mobile, 2015b] Vision Mobile, IoT Megatrends 2016 – Six key trends in the IoT developer economy, 2015, http://www.visionmobile.com/product/iot-megatrends-2016/

[VMTS] Visual Modeling and Transformation System, http://www.aut.bme.hu/vmts/

[Williams and Bainbridge, 1988] T. Williams and B. Bainbridge, Rule based systems, In Approaches to knowledge representation: an introduction, Research Studies Press Ltd., Taunton, UK, pp. 101-115, 1988.

[Xiao et al, 2010] Y. Xiao, M. Siekkinen and A. Yla-Jaaski, Framework for energy-aware lossless compression in mobile services: The case of e-mail, IEEE International Conference on Communications, pp. 1-6, 2010.