GI-Edition

Lecture Notes in Informatics

Andreas Oberweis, Ralf Reussner (Hrsg.)

Modellierung 2016

2.–4. März 2016
Karlsruhe

Proceedings

254

A. Oberweis, R. Reussner (Hrsg.): Modellierung 2016

Andreas Oberweis,
Ralf Reussner (Hrsg.)

Modellierung 2016

2. – 4. März 2016
Karlsruhe, Deutschland

Gesellschaft für Informatik e.V. (GI)

Lecture Notes in Informatics (LNI) – Proceedings
Series of the Gesellschaft für Informatik (GI)

Volume P-254

ISBN978-3-88579-648-0 ISSN 1617-5468

Volume Editors

Prof. Dr. Andreas Oberweis
Dr. Stefanie Betz
PD Dr. Agnes Koschmider
Karlsruher Institut für Technologie
Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB)
Kaiserstraße 89, 76133 Karlsruhe, Deutschland
E-Mail: oberweis@kit.edu, stefanie.betz@kit.edu, agnes.koschmider@kit.edu

Prof. Dr. Ralf Reussner
Dr.-Ing. Erik Burger
Dr. rer. nat. Robert Heinrich
Karlsruher Institut für Technologie
Institut für Programmstrukturen und Datenorganisation (IPD)
Am Fasanengarten 5, 76131 Karlsruhe, Deutschland
E-Mail: reussner@kit.edu, burger@kit.edu, heinrich@kit.edu

Series Editorial Board

printed by Köllen Druck+Verlag GmbH, Bonn

Vorwort

Modelle stellen ein der wichtiges Hilfsmittel zur Beherrschung komplexer Systeme dar. Die Themenbereiche der Entwicklung, Nutzung, Kommunikation und Verarbeitung von Modellen sind so vielfältig wie die Informatik mit all ihren Anwendungen.

Die Fachtagung „Modellierung“ wird vom Querschnittsfachausschuss Modellierung der Gesellschaft für Informatik e.V. seit 1998 durchgeführt und hat sich als einschlägiges Forum für Grundlagen, Methoden, Techniken, Werkzeuge sowie Domänen und Anwendungen der Modellierung etabliert. Die „Modellierung“ führt Teilnehmerinnen und Teilnehmer aus allen Bereichen der Informatik sowie aus Wissenschaft und Praxis zusammen. Die Tagung zeichnet sich traditionell durch lebendige und fachgebietsübergreifende Diskussionen aus, weshalb sie gerade auch für Nachwuchswissenschaftlerinnen und Nachwuchswissenschaftler interessant ist.

Der vorliegende Tagungsband erhält 17 Beiträge. Es wurden insgesamt 31 Beiträge eingereicht, wovon 11 Beiträge als Vollbeiträge und 6 Beiträge als Kurzbeiträge ausgewählt wurden. Die Begutachtung erfolgte durch das Programmkomitee der Modellierung 2016. Während des Auswahlprozesses bestand für die Autorinnen und Autoren die Möglichkeit, zu ihren Gutachten Stellung zu nehmen. Die angenommenen Beiträge behandeln aktuelle Erkenntnisse zu Grundlagen, Methoden, Techniken und Werkzeugen der Modellierung. Im Speziellen werden Arbeiten zur Modellbildung, Modellierungssprachen, Modelltransformation, Modellierungstechniken, Modellvalidierung, sowie zum modellbasierten Testen vorgestellt.

Für ihre Beteiligung an der Modellierung 2016 möchten wir uns bei allen Autorinnen und Autoren bedanken. Den Mitgliedern des Programmkomitees danken wir für die sorgfältige und termingerecht Begutachtung, Bei Prof. Gerti Kappel, Dr. Oliver Raabe und Dr. Thomas Karle bedanken wird uns für Ihre Keynote-Vorträge. Unser weiterer Dank gilt den vielen lokalen Helfern für die Vorbereitung und Durchführung der Veranstaltung. Den Firmen PROMATIS Software GmbH, andrena objects, PPI Akteingesellschaft und TWT danken wir für die finanzielle Unterstützung.

Karlsruhe, im März 2016

Stefanie Betz
Erik Burger
Robert Heinrich
Agnes Koschmider
Andreas Oberweis
Ralf Reussner

Sponsoren

Andrena
www.andrena.de

Promatis
www.promatis.de

PPI AG
www.ppi.de

TWT GmbH
www.twt-gmbh.de

Partner

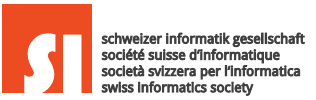
Karlsruher Institut für Technologie
www.kit.edu

FZI Forschungszentrum Karlsruhe
www.fzi.de

Gesellschaft für Informatik
www.gi.de

Österreichische Computer Gesellschaft
www.ocg.at

Schweizer Informatik Gesellschaft
www.s-i.ch



Tagungsleitung

Gesamtleitung

Andreas Oberweis (KIT/FZI, Karlsruhe), Ralf Reussner (KIT/FZI, Karlsruhe)

Workshops

Stefanie Betz (KIT, Karlsruhe), Ulrich Reimer (FHS St. Gallen)

Praxisforum

Dr. Judith Michael (Universität Klagenfurt), Alexander Paar (TWT GmbH), Christian Stahl (TWT GmbH)

Tutorien

Erik Burger (KIT, Karlsruhe), Norbert Seyff (Universität Zürich)

Werkzeugpräsentation

Hans-Georg Fill (Universität Wien), Agnes Koschmider (KIT, Karlsruhe)

Studierendenprogramm

Robert Heinrich (KIT, Karlsruhe), Rainer Neumann (Hochschule Karlsruhe)

Programmkomitee

Stefanie Betz, Karlsruher Institut für Technologie
Ruth Breu, Universität Innsbruck
Erik Burger, Karlsruher Institut für Technologie
Gregor Engels, Universität Paderborn
Hans-Georg Fill, Universität Wien
Ulrich Frank, Universität Duisburg-Essen
Holger Hermanns, Universität des Saarlandes
Georg Hinkel, Karlsruher Institut für Technologie
Jan Jürjens, TU Dortmund
Dimitris Karagiannis, Universität Wien
Agnes Koschmider, Karlsruher Institut für Technologie
Thomas Kuehne, Victoria University of Wellington
Florian Matthes, Technische Universität München
Heinrich C. Mayr, Alpen-Adria-Universität Klagenfurt
Günther Müller-Luschnat, iteratec GmbH
Friederike Nickl, Swiss Life

Barbara Paech, Universität Heidelberg
Ralf Reussner, Karlsruher Institut für Technologie
Matthias Riebisch, Universität Hamburg
Bernhard Rumpe, RWTH Aachen
Ina Schaefer, Technische Universität Braunschweig
Andreas Schoknecht, Karlsruher Institut für Technologie
Andy Schürr, TU Darmstadt
Friedrich Steimann, Fernuniversität in Hagen
Peter Tabeling, INTERVISTA AG
Bernhard Thalheim, Christian-Albrechts-Universität zu Kiel
Meike Ullrich, Karlsruher Institut für Technologie
Mathias Weske, HPI, Universität Potsdam
Heinz Züllighoven, Universität Hamburg

Querschnittsfachausschuss Modellierung

Die Modellierung 2016 ist eine Arbeitstagung des Querschnittsfachausschusses Modellierung (www.gi-modellierung.de), in dem folgende GI-Fachgliederungen vertreten sind:

ARC (Architekturen)

ASE (Automotive Software Engineering)

EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

FoMSESS (Formale Methoden und Software Engineering für Sichere Systeme)

ILLS (Intelligente Lehr- und Lernsysteme)

MMB (Messung, Modellierung und Bewertung von Rechensystemen)

MobIS (Modellierung betrieblicher Informationssysteme)

PN (Petrinetze)

RE (Requirements Engineering)

ST (Softwaretechnik)

WI-VM (Vorgehensmodelle für die betriebliche Anwendungsentwicklung)

WM (Wissensmanagement)

Inhalt

Modellbildung

Modellierungssprachen

Modelltransformationen (Kurzbeiträge)

Modellierungstechniken

Modellbasiertes Testen und Modellvalidierung

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 13

Pushing the CIDOC-Conceptual Reference Model towards Linked Open Data by Open Annotations

Matthias Frank[1], Stefan Zander[1]

Abstract: By using a novel modelling approach, we demonstrate how the Conceptual Reference Model (CRM) of ICOM's International Committee for Documentation (CIDOC) can be complemented using the Open Annotation Data Model (OADM) in order to create semantically rich annotations. We show that domain knowledge can be combined with meaningful and linked data exposed in the so-called Web of Data (aka semantic Web) by having the necessary provenance information for annotations. The combination of domain specific knowledge with existing Linked Open Data (LOD) requires well-designed modelling decisions for linking semantic data sets in a comprehensible way. We show that our combined approach is able to address the requirements of digital heritage in more sufficient ways than each model separately. We combine the advantages of a proven domain ontology with the flexibility and semantic richness of the OADM. In order to evaluate our approach, we show with a concrete example how a museum artifact is modeled in CIDOC-CRM and how these data can be interlinked with existing LOD in meaningful and machine-processable ways by encoding provenance information for new annotations using the OADM.

Keywords: Semantic Web, CIDOC-CRM, Open Annotation Data Model, Linked Open Data

1 Introduction

The digitization of our cultural heritage (CH), also known as digital heritage (DH), is one of the big challenges museums all over the world are faced with [KK13]. Therefore, the United Nations Educational, Scientific and Cultural Organization (UNESCO) has developed the *Charter on the Preservation of Digital Heritage* [Un04] to provide best practice guidelines for preserving DH. As stated by the UNESCO, DH "is inherently unlimited by time, geography, culture or format", which requires advanced data modeling approaches.

One approach to model DH data is the Conceptual Reference Model (CRM) [CI13] introduced by the ICOM's International Committee for Documentation (CIDOC)[2]. As of today, a growing number of museums and DH projects like the British Museum[3], the Smithsonian American Art Museum (SAAM)[4] or the Classical Art Research Online Services[5] have started to publish semantically enriched data about their hosted objects using the CIDOC-CRM. However, from a data consuming point of view, modelling DH in this way has a limitation: The CIDOC-CRM does not provide means to encode provenance information

[1] FZI Forschungszentrum Informatik am KIT, Information Process Engineering, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, {frank, zander}@fzi.de
[2] http://network.icom.museum/cidoc
[3] http://www.britishmuseum.org/
[4] http://americanart.si.edu/
[5] http://www.clarosnet.org

for the annotations itself. Therefore, when using DH data modeled in CIDOC-CRM in an application, it is impossible to deduce implicit information like the trustworthiness or comparability of annotations. Without this information, data integration in cross-domain projects and the reuse and interpretation of annotations in different contexts is hardly possible.

In order to overcome these limitations of CIDOC-CRM, we introduce a novel modeling approach that extends CIDOC-CRM by combining its well-structured and proven domain-specific taxonomy with the flexibility of rich annotations using the Open Annotation Data Model (OADM) [SCV13].

Representing DH annotations using a combination of CIDOC-CRM and OADM encourage the following:

1. *Interoperability*: As the OADM is a draft of the World Wide Web Consortium (W3C) community, OADM-annotations can be consumed and interpreted correctly by varying applications.
2. *Adaption*: Semantically rich annotations like OADM-annotations allow to adapt the meta data contained within an annotation without cutting the link between body and target of the annotation.
3. *Reuse and interlinkage*: Once a semantically rich annotation is created, it can be reused by adding further bodies or targets to this annotation.
4. *Trustworthiness*: By stating the authorship of an annotation and add explanatory statements to it, annotations become reproducible and the trustworthiness of annotations can be justified.

As a consequence, the main outcome of this work is to provide the logical underpinning upon which CIDOC-CRM can be beneficially extended using the OADM in creating semantically rich annotations that can be shared, extended and utilized by related approaches and also adopted by other domains, e.g. incorporated in the *Educational Web of Data* [Br11].

2 Background

In this section we provide an overview of the current modelling approaches for CH and Linked Open Data (LOD). The focus is on CIDOC-CRM, OADM and LOD in general.

2.1 CIDOC-CRM

Museums as a stakeholder in the process of digital preservation of artifact descriptions perform archival functions, like building and maintaining reliable collections of well-defined digital objects. They preserve the features like content, fixity, reference, provenance and context which give these objects their integrity. To keep the integrity for digital information objects with long-term cultural value intact is a precondition in order to use them for referring, indexing, citing or any other purpose by the consumers of that data. [LT01, p. 47] For modeling ontologies in the knowledge domain of CH, CIDOC has defined a reference model for storing these digital objects: The CIDOC-Conceptual Reference Model. CIDOC-CRM defines the Terminological Box (TBox) for ontologies in the domain of CH that covers all concepts relevant to describe all types of material collected and displayed by museums and related institutions[BCT07, p. 255]. This terminology is also applied for Lightweight Information Describing Objects (LIDO), an Extensible Markup Language (XML) schema for CH provided by International Council of Museums (ICOM)[IC10]. An example for modeling DH data in CIDOC-CRM is shown in Figure 1. The marked nodes in Figure 1 represent the classes defined in CIDOC-CRM.

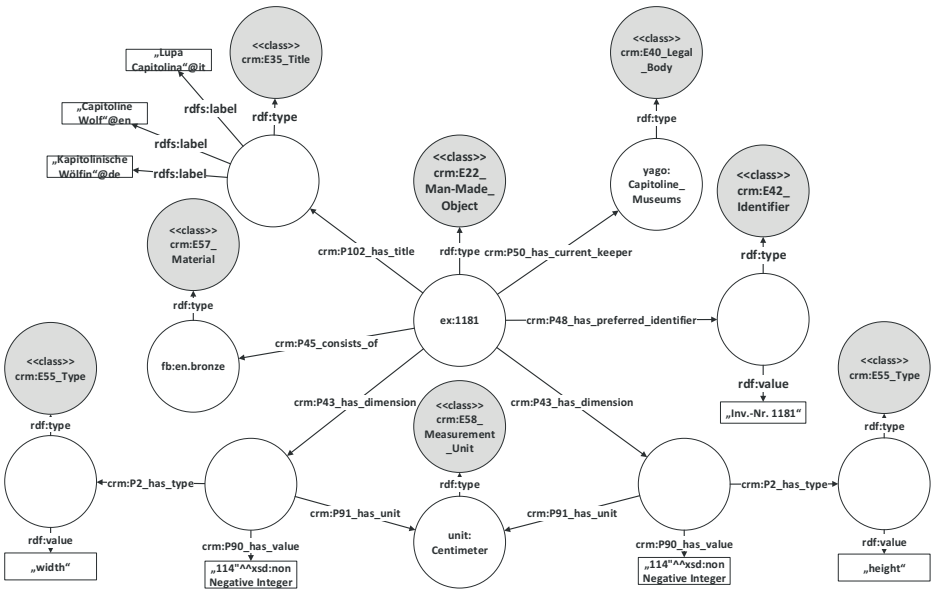


Fig. 1: Example for modelling in CIDOC-CRM

2.2 Open Annotation Data Model

Annotations created with Web Ontology Language (OWL)/Resource Description Framework (RDF) have a limited expressiveness by default. Basically, these annotations state that two resources are related to each other in a specific way. This relationship may be a predicate like `rdfs:seeAlso`, `rdfs:isDefinedBy` or any other predicate that expresses a relation of these resources. Within the scope of this work, the subject of the triple expressing the annotation is named *body* of the annotation, whereas the object of the triple is named *target* of the annotation. Such a basic annotation is depicted in Figure 2.

Fig. 2: Basic RDF Annotation

As there has been no uniform approach for creating annotations, in 2012 the W3C community introduced the OADM. This approach introduces a methodology for annotations that conforms to the architecture of the World Wide Web (WWW)[CSV12]. The OADM consists of a core which provides the basic functionality to create open annotations. The OADM core can be extended by several modules if necessary for a specific application. Rather than implementing an annotation as a simple triple pointing from a `body` resource to a related `target`, OADM creates a distinct resource for the annotation itself which then points to the `body` and the `target` of the annotation and also provides useful metadata. The idea of OADM is to reuse existing vocabulary wherever possible, for example the vocabulary defined in Friend of a Friend (FOAF) or Dublin Core (DC). The reuse of LOD-resources is therefore a major contribution of our approach. However, there are also classes and properties defined for the open annotation namespace in `http://www.w3.org/ns/oa#`, usually abbreviated with the prefix `oa:`. A depiction of the basic annotation model is shown in Figure 3, stating the same fact as in Figure 2.

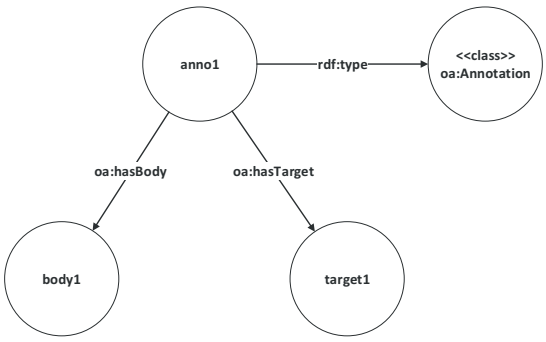


Fig. 3: Open Annotation Data Model

Metadata that can be included by OADM covers a person or organization that created the annotation, the motivation for the annotation and also the time when the annotation was created. This approach allows to express the authorship of an annotation and separate between annotations created by the museum's staff and annotations created by the community. In addition, OADM provides the possibility to state the software that was used for serialization and the time when the annotation was serialized, both can be taken into consideration when justifying the provenance and trustworthiness of an annotation. Especially the time of serialization is important for a proper version control [SCV13] when maintaining the annotations. With OADM, conservators of CH have the option to use linked data to augment user experience rather than only publishing their own linked open datasets. The OADM is therefore ideally suited for the creation of knowledge structures through semantic annotations in the field of CH.

2.3 Linked Open Data

The idea of linked data was introduced by Tim Berners-Lee in 2006 [BL06]. He defined expectations which apply to both, the conventional Web of Hypertext Markup Language (HTML) documents and the Web of linked data represented using the RDF. He also added a 5-Star scheme for LOD in 2010.

An early use of publishing linked data is the creation and publishing of personal profiles as some kind of business cards using the FOAF vocabulary. These FOAF-profiles have made an essential part of linked data in the early beginning of the semantic Web. However, the number of datasets in linked data, the number of triples within these datasets and also the RDF links interlinking these datasets have increased rapidly during the last years. Richard Cyganiak and Anja Jentzsch started an approach to visualize datasets of linked data available on `http://datahub.io/dataset?tags=lod` as a LOD cloud diagram in 2007. The first version of this diagram contained 12 datasets. This project was maintained over the years, in 2014 the LOD cloud diagram contained 570 datasets. 374 of them were described by the data providers themselves in the datahub.io dataset catalog, 196 more were discovered by a crawl of the Linked Data web conducted in April 2014 [CJ14]. A depiction of the resulting LOD cloud diagram is shown in Figure 4.

As can be seen from this diagram, FOAF-profiles, Geo-Names and especially DBpedia are highly interconnected datasets containing a huge amount of triples. For example, a SPARQL Protocol and RDF Query Language (SPARQL) query counting the triples contained in DBpedia in January 2015 returns a number of more than two billion triples.

3 Related Works

For the creation of annotations for museum objects in this work, we annotate the description of a museum artifact modeled in CIDOC-CRM with data from the LOD cloud. According to a research funded by SAAM, mapping the data of a museum to linked data involves three steps [Sz13, p. 1–2]:

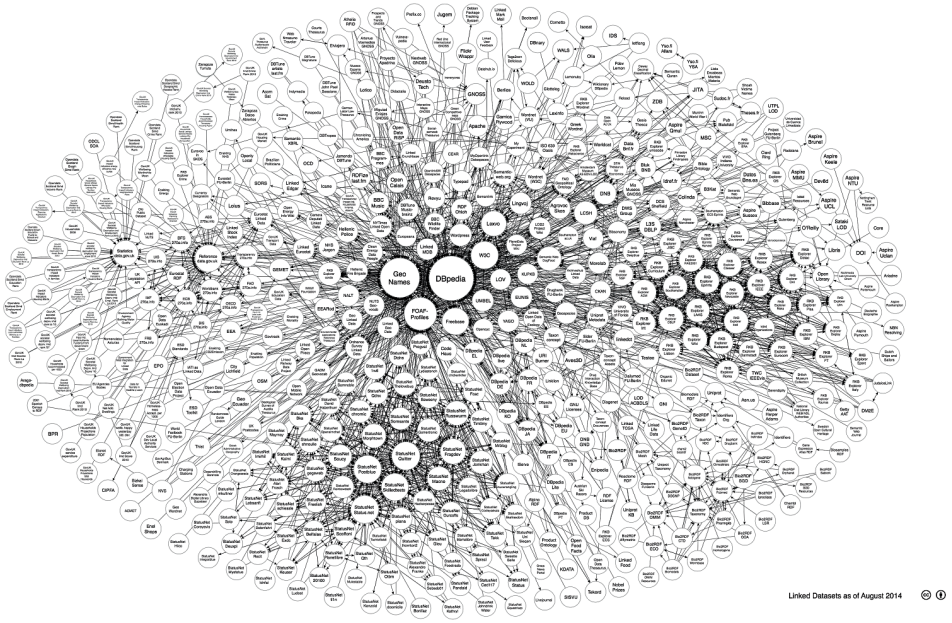


Fig. 4: Linked Open Data Cloud as of 2014

1. Map the Data to RDF.
   The first step is to map the metadata about works of art into RDF. This involves selecting or writing a domain ontology with standard terminology for works of art and converting the data to RDF according to this ontology. De Boer et al.[6] note that the process is complicated because many museums have rich, hierarchical or graph-structured data. The data often includes attributes that are unique to a particular museum, and the data is often inconsistent and noisy because it has been maintained over a long period of time by many individuals. In past work, the mapping is typically defined using manually written rules or programs.

2. Link to External Sources
   Once the data is in RDF, the next step is to find the links from the metadata to other repositories, such as DBpedia or GeoNames. In previous work, this has been done by defining a set of rules for performing the mapping. Because the problem is difficult, the number of links in past work is actually quite small as a percentage of the total set of objects that have been published.

[6] Boer, V., Wielemaker, J., Gent, J., Hildebrand, M., Isaac, A., Ossenbruggen, J., Schreiber, G.: Supporting Linked Data Production for Cultural Heritage Institutes: The Amsterdam Museum Case Study. In: Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V. (eds.) Lecture Notes in Computer Science, pp. 733-747. Springer Berlin Heidelberg (2012), cited by [Sz13]

3. Curate the Linked Data
   The third step is to curate the data to ensure that both the published information and its links to other sources within the LOD are accurate. Because curation is so labor intensive, this step has been largely ignored in previous work and as a result links are often inaccurate.

We do also consider these three steps for our modeling approach of semantic annotations in the field of CH. As stated in the outcome of the SAAM project, for the first step there are already some successful approaches like the Europeana project[7], the Amsterdam Museum[8], the LODAC museum[9] or the KARMA approach[10]. Therefore, our work focuses on the linking of museums' RDF-data to external resources and also in maintaining these links. For structuring metadata of CH objects we extend the domain specific model with the OADM. This idea was also suggested for future work for structuring metadata of CH objects during the International Conference in DC and Metadata Applications in 2014 [Wi14].

For this work, we assume that data of a museum is modeled in CIDOC-CRM in order to describe museum objects. The goal is to ensure that on the one hand only the museum itself is able to publish authoritative data, but on the other hand the community is able to create annotations to this data in order to augment the knowledge structure of the museum's repository.

4 Approach

With our approach we show that the combination of the domain specific modeling of CIDOC-CRM and the OADM leads to an comprehensive model that covers the requirements of CH specialists and enables the modeling of provenance information for new annotations. This novel approach helps museums to use LOD in order to augment their visitors experience and also publish their data as LOD in a meaningful way by providing provenance data of annotations, which allows a collaborative annotation of museum objects.

4.1 Requirements

In order to contribute to the process of digital preservation of CH, we pose the following requirements:

R1 The data published by a museum about their artifacts has to be modeled in a way that consumers can distinguish them from annotations created by the user community.

[7] http://data.europeana.eu
[8] http://www.amsterdammuseum.nl/open-data
[9] http://www.ontotext.com/customers/lodac-museum-linked-open-data-academia/
[10] http://www.isi.edu/integration/karma/

R2 The user-created annotations have to be serialized and published in order to augment user experience when consuming DH-objects.

R3 Methods that enables users to filter annotations by type, creator, annotated object, annotation time and version has to be provided.

All three requirements are on the approach in general, rather than for a specific prototype. The requirements are the result of our practical research work with partners from the CH-domain. To fulfill requirement R1, museums' data is published in CIDOC-CRM in read-only while annotations by users are added using OADM in order to clearly state the provenance of the created annotations. As an example, when the conservator of a museum has gathered the required information and added them to the system, any user can search for resources with similar properties on the LOD cloud and create personal annotations. Due to the use of the OADM, not only the semantic annotation itself will be created, but also some useful meta data. This meta data may include information about who created this annotation, what was his or her motivation doing that, when he or she actually created the annotation and also which software was used for the serialization and when was the serialization of the annotation performed.

4.2 Data Structure for Museum Objects

The description of museum objects is assumed to be modeled in CIDOC-CRM for this work. However, as the full CRM aims to define all classes and properties needed to describe knowledge in the domain of CH and not only museum objects, for this work only instances of the class `E22 Man-Made Object` defined by CIDOC in the CRM are considered. This comprises physical objects purposely created by human activity [CI13, p. 11] as artifacts of CH. This Section describes the properties which are applicable to the class `E22 Man-Made Object` and introduces the exemplary modeling of the description of a museum object as an instance of this class.

Figure 5 shows the Unified Modeling Language (UML) class diagram of the TBox for the class `E22 Man-Made Object`. The members of the classes are the respective properties of the classes, consisting of predicate (identifier `P`) and object (identifier `E`). In RDF, properties are not added to a class like in an object orientated modeling approach. In fact, the properties shown in Figure 5 result from the domain and range defined for each predicate. Although the class `E22 Man-Made Object` does not have any specified properties, it inherits all the properties from its superclasses.

4.3 Discover Resources in Linked Data for new Annotations

The core of our approach is to support the conservator of a museum in creating rich annotations for digitized artifacts modeled in CIDOC-CRM with resources available as LOD. This is done by providing suggestions for annotations and enable volunteers to review these suggestions.

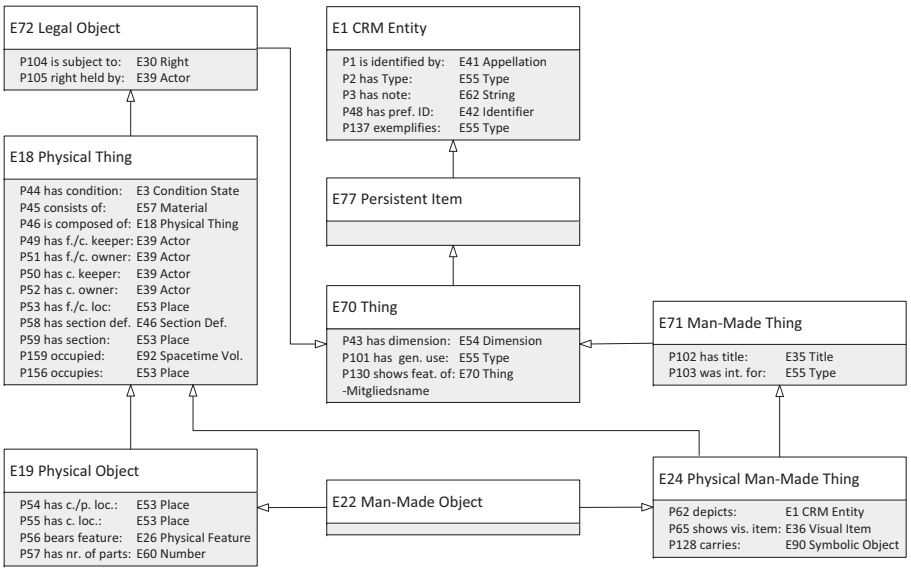


Fig. 5: UML class diagram for E22 Man-Made Object with superclasses

In order to find suitable resources for the semantic annotation of digitized artifacts, all resources which are related to the subject and are available as LOD should be discovered. Relatedness of resources in the semantic Web can be measured in different ways, for example the number of edges between two nodes. This approach does not consider the quality of the edges, therefore each predicate of an RDF-graph is treated similar. There are also approaches that weight the quality of edges, which means that a even a path with more edges could express a higher relatedness between two nodes if the quality of the edges is better, or in sense of RDF the predicates are semantically better suited to express a relation. In the context of our work, relatedness is measured in the number and quality of property matches, as a high accordance of properties does also indicate a high relatedness. Museum objects are objects of public interest, therefore it is likely that someone already published something with similar properties as LOD.

However, as the semantic Web is not a central database where properties are always defined in the same way, there may be relevant resources which are described with different properties which have the same meaning. To find these resources, the predicates have to be mapped to cover these different descriptions. An abstracted depiction of this mapping is shown in Figure 6.

Rather than just search for resources with similar properties of the subject in the example data introduced in Section 5.1, like for example the property `crm:P45_consists_of fb:en.bronze`, both, the predicate and the object have to be mapped to lists of equivalent predicates and objects. When the input data is enriched by lists of equivalent predicates and objects, resources with similar properties published as LOD can be queried. This is

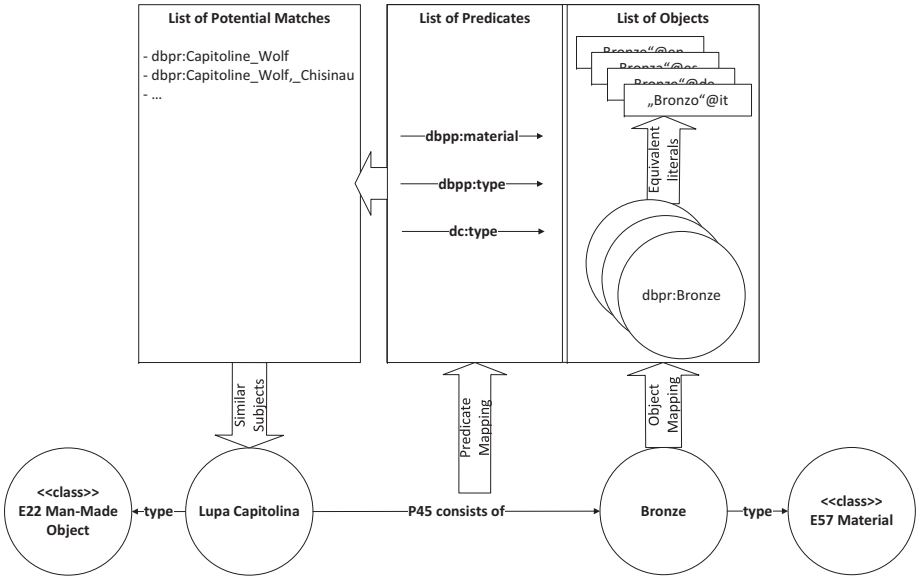


Fig. 6: Abstracted Mapping Process

done with the help of SPARQL using public SPARQL endpoints. The result of this query is then returned to the client for further processing.

The resulting graph of this new annotation is shown in Figure 7. However, in this graph, target and body of the annotation are not shown as actual resources. The abstracted body shows that the Internationalized Resource Identifier (IRI) resources used for the body of the annotation come from within the LOD cloud, whereas the abstracted target shows that the IRI-resource used for the target of the annotation is part of the museum's repository modeled in CIDOC-CRM. By using OADM for user created annotations, R2 and R3 are fulfilled.

The OADM annotations created by our approach combine the digital description of CH artifacts modeled with CIDOC-CRM with information available as LOD. The result is therefore a knowledge structure that contains information of CH from both sources, including metadata about the annotations themselves. Figure 8 shows an example of an artifact modeled in CIDOC-CRM, the marked nodes represent an additional annotation about this artifact modeled in OADM.

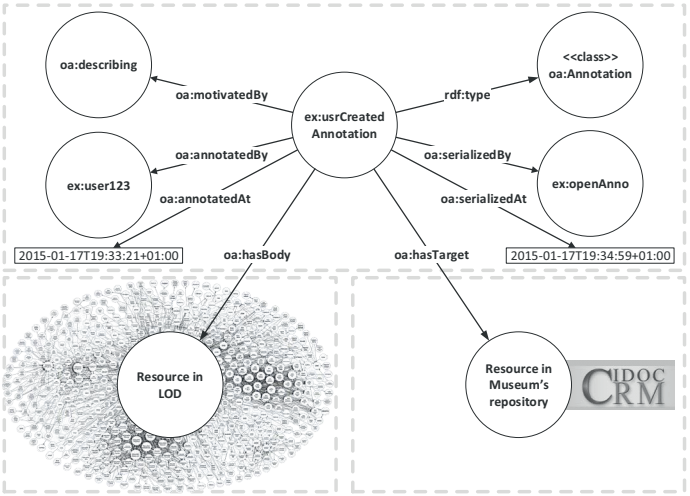


Fig. 7: RDF-Graph of OADM-Annotation

5 Use Case – Evaluation

5.1 Example data: Lupa Capitolina

Our approach presumes valid CIDOC-CRM ontology data. Therefore, in addition to the schema description of `E22 Man-Made Object` (TBox) given in Section 4.2, some instance data (Assertional Box (ABox)) has to be added. In our example, we assume that the Capitoline Museums in Rome, Italy, wants to publish a semantic descriptions of their hosted artifacts using CIDOC-CRM. First, the museum defines `http://museum.example.com/objects/` as the namespace for all museum objects. This namespace will be abbreviated with the prefix `museum:`. The museum artifact that is encoded in this example is the bronze sculpture "Capitoline Wolf" (Italian: Lupa Capitolina). A description of this sculpture is provided by the university of cologne[11]. The preferred local name for an artifact within the museums namespace is the inventory number of the corresponding object [CI11]. In case of the Capitoline Wolf, the Capitoline Museums assigned the inventory number `1181`. The resulting IRI for the new object is therefore `http://museum.example.com/objects/1181`, abbreviated as `museum:1181`. This abbreviation is not a valid qualified name (QName) as the local name starts with a number, however, it is a valid Compact URI expression (CURIE).

As CIDOC-CRM does not foresee the description of instances of `E58 Measurement Unit` [CI13, p. 23], instances provided by the Quantities, Units, Dimensions and Data Types Ontologies (QUDT) are used. The namespace for this ontology is `http://qudt.org/vocab/unit#` (abbreviated with `unit:`). QUDT is developed by TopQuadrant and

[11] `http://arachne.uni-koeln.de/item/objekt/16611`

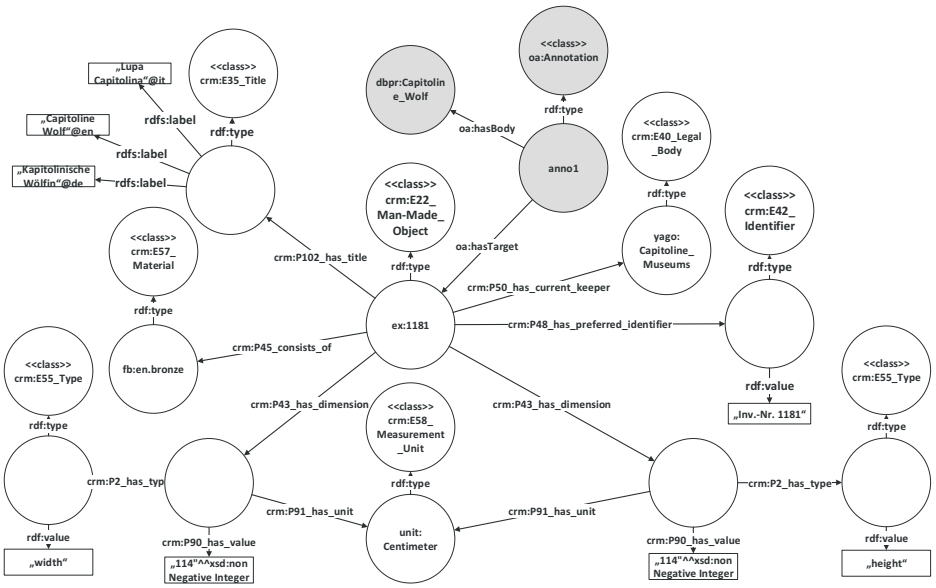


Fig. 8: CIDOC-CRM extended by OADM

the National Aeronautics and Space Administration (NASA) in order to provide interoperability between information systems. It is published under Creative Commons (CC) Attribution-Share Alike 3.0 United States License and can therefore be freely used, as long as the name of the creator is provided. For instances of other classes of CIDOC-CRM, for example `E39 Actor` or `E53 Place`, existing upper ontologies are used where applicable. In order to improve the reuse of resources of the semantic Web, we use upper ontologies which are published with an open license. In this example, we use the upper ontologies yago[12] (abbreviated with `yago:`) and freebase[13] (abbreviated with `fb:`).

5.2 Related Resources in LOD for new Annotations of Lupa Capitolina

The discovery of resources in LOD that are related to a fact which is described with a blank node (bnode) in the museum's repository requires additional attention. Although it is possible to query bnodes due to the graph-oriented semantics of SPARQL, many subjects in LOD are annotated in a more simple way. As an example, the height of the artifact used here and modeled in CIDOC-CRM is stated with an dimension-bnode having the local Identifier (ID) `_:lh`. This dimension has the type "height" and a value of 75 cm. The same fact is expressed in DBpedia[14], a database that contains structured data from Wikipedia, with the properties `dbpprop:heightMetric "75"` and `dbpprop:metricUnit "cm"`.

[12] `http://yago-knowledge.org/resource/`
[13] `http://rdf.freebase.com/ns/`
[14] `http://dbpedia.org/resource/Capitoline_Wolf`

Therefore, all properties modeled as a bnode have to be interpreted by the properties of the respective bnode. A first-degree (one edge distance from the subject) bnode-property may have bnode-properties as well. The same goes for the second-degree (distance of two edges), third-degree (distance of three edges) and so on, therefore this recursive procedure has to be limited to a particular distance level in order to answer queries in a reasonable time. A limit of 3 runs in example does ensure to get all properties up to a distance of three edges.

Our approach is implemented as a prototype which searches LOD for related resources to annotate a given CH artifact. When the search has finished, all results are listed, aggregated by instances and ordered by the number of matches as can be seen in Figure 9. The resource found with the most according properties is listed first. All results that fulfill the predefined requirements are preselected. For the prototypical implementation used for our work, the default parameters are $P_1 = 2$ for the minimum number of congruent properties and $P_{1=0.2}$ for the minimum rate of congruent properties in relation to the total number of properties. Therefore, all resources which have at least two properties that matches to properties of the target resource and the number of matching properties in relation to the total number of properties of the target resource equals to at least 20% are preselected for annotation. In this example, three resources are suggested to be annotated with `ind:Lupa_Capitolina` as can also be seen in Figure 9.

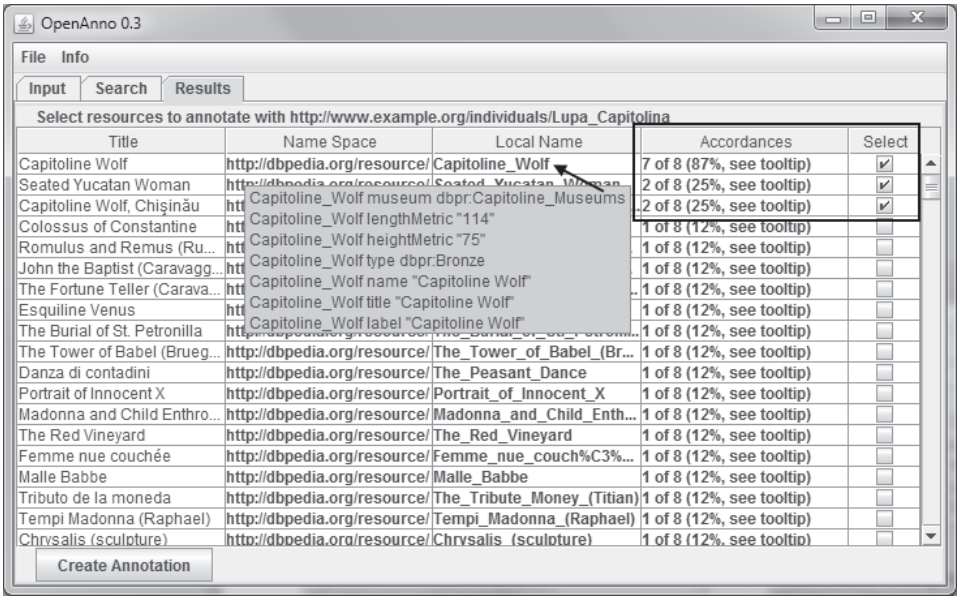


Fig. 9: Positive matches

Ideally, all preselected resources are related to the target and therefore suited for an annotation. To check whether the suggestion does apply, a tooltip does show up when moving the cursor to the number of according properties of that resource. The tooltip indicates which properties exactly are in accordance with the target. Based on this information the user can decide whether this resource should be annotated or not. One example of a preselected

resource is shown in Figure 9. This resource is preselected as the number of properties that match properties of the target resource fulfill the requirements $7 \geq 1$ and $\frac{7}{8} \geq 0.2$. The tooltip of Figure 9 indicates that this resource is in fact related to the resource introduced in Section 5.1. Therefore, the suggestion of our prototype was correct (true positive) in this case.

5.3 Creating new Annotations

Once the resources for annotation are reviewed, the annotation can be serialized. When serialized, an info message is displayed to show the result of the serialization including the ID for the newly created annotation, as shown in Figure 10. In order to fulfill R2, this serialization has also to be published. The provenance information of the new annotation encoded in this output can be used to filter annotations by type, creator, annotated object, annotation and version, which fulfills R3.

```
@prefix prov:  <http://www.w3.org/ns/prov#> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix agent: <http://www.example.org/agents/> .
@prefix dbpr:  <http://dbpedia.org/resource/> .
@prefix oa:    <http://www.w3.org/ns/oa#> .
@prefix ind:   <http://www.example.org/individuals/> .
@prefix anno:  <http://www.example.org/annotations/> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

anno:88e6667b-00f0-4591-89a9-618481c4f13a
        a                oa:Annotation ;
        oa:annotatedAt   "2015-01-27T16:02:03.309Z"^^xsd:dateTime ;
        oa:annotatedBy   agent:e98eb2d5-fd26-4b29-9ab5-dddaed08c12f ;
        oa:hasBody       dbpr:Capitoline_Wolf ,
                         dbpr:Capitoline_Wolf,_Chisinau ;
        oa:hasTarget     ind:1181 ;
        oa:motivatedBy   oa:editing ;
        oa:serializedAt  "2015-01-27T16:02:34.213Z"^^xsd:dateTime ;
        oa:serializedBy  agent:a69f0971-ff45-4af0-a197-a3f61cfa163d .

agent:e98eb2d5-fd26-4b29-9ab5-dddaed08c12f
        rdf:Type   foaf:Organization ;
        foaf:name  "FZI" .

agent:a69f0971-ff45-4af0-a197-a3f61cfa163d
        rdf:Type   prov:SoftwareAgent ;
        foaf:name  "OpenAnno 0.3" .
```

Fig. 10: Output of OpenAnno in Turtle

6 Limitations and Conclusion

In this paper, we have shown that a domain ontology can be extended by OADM in order to provide meaningful, rich annotations. Our approach allows to include provenance information for new annotations for data modeled in a domain ontology without destroying the structure of the domain ontology. By including provenance information of annotations the annotation process may also be crowdsourced as an collaborative task for new annotations without decreasing the quality of museums' data. We have shown how this approach can be used in order to complement DH data by annotating them with existing resources in LOD while obtaining the provenance information of the new annotations.

However, there are limitations for our approach. In particular, when using the OADM for annotations, it is not possible to state the type of relation between the annotated resource and the resource used for the annotation explicitly, e.g. "is part of", "consists of" or "is same as". These specific relations have to be implemented separately, as they are not provided by the OADM. In addition, our approach does not include any cryptography that ensures the authenticity of the encoded provenance information. For a real justification of the trustworthiness of annotations, a cryptography module has to be added in order to ensure the authenticity of annotations.

Acknowledgements. This work was supported by the German Federal Ministry for Economic Affairs and Energy (BMWI) within the CultLab3D project (Ref. 01MT12022D).

References

[BCT07] Breitman, K. K.; Casanova, Marco Antonio; Truszkowski, Walt: Semantic Web: Concepts, technologies and applications. NASA monographs in systems and software engineering. Springer, New York and London, 2007.

[BL06] Berners-Lee, Tim: , Linked Data - Design Issues. `http://www.w3.org/DesignIssues/LinkedData.html`, 2006.

[Br11] Bratsas, Charalampos; Dimou, Anastasia; Alexiadis, Georgios; Kavargyris, Konstantinos; Parapontis, Ioannis; Bamidis, Panagiotis; Antoniou, Ioannis: , Educational Semantic Wikis in the Linked Data Age: The case of MSc Web Science Program at Aristotle University of Thessaloniki. Linked Learning 2011: the 1st International Workshop on eLearning Approaches for the Linked Data Age, 2011.

[CI11] CIDOC: , CIDOC-ICOM recommendation on Linked Open Data for museums –Draft. `http://cidoc-crm.org/docs/LoD_For_Museums_v1.7-en.doc`, 2011.

[CI13] CIDOC CRM Special Interest Group: , Definition of the CIDOC Conceptual Reference Model. `http://cidoc-crm.org/docs/cidoc_crm_version_5.1.2.pdf`, 2013.

[CJ14] Cyganiak, Richard; Jentzsch, Anja: , The Linking Open Data cloud diagram. `http://lod-cloud.net/`, 2014.

[CSV12] Ciccarese, Paolo; Sanderson, Robert; Van de Sompel, Herbert: , Open Annotation Draft Data Model. `http://www.openannotation.org/spec/core/20120328.html`, 2012.

[IC10] ICOM: , LIDO's background. `http://network.icom.museum/cidoc/working-groups/lido/lido-overview/lidos-background/`, 2010.

[KK13] Klimpel, Paul; Keiper, Jürgen, eds. Was bleibt? Nachhaltigkeit der Kultur in der digitalen Welt. iRights.Media, Berlin, 2013.

[LT01] Lazinger, Susan S.; Tibbo, Helen R.: Digital preservation and metadata: History, theory, practice. Libraries Unlimited, Englewood, Colo., 2001.

[SCV13] Sanderson, Robert; Ciccarese, Paolo; Van de Sompel, Herbert: , Open Annotation Data Model. `http://www.openannotation.org/spec/core/`, 2013.

[Sz13] Szekely, Pedro; Knoblock, Craig A.; Yang, Fengyu; Zhu, Xuming; Fink, Eleanor E.; Allen, Rachel; Goodlander, Georgina: Connecting the Smithsonian American Art Museum to the Linked Data Cloud. 2013.

[Un04] United Nations Educational, Scientific and Cultural Organization: Records of the General Conference: 32nd Session. United Nations Educational, Scientific and Cultural Organization, Paris, 2004.

[Wi14] Wittenberg, Jamie: Retaining Metadata in Remixed Cultural Heritage Objects. Proc. Int'l Conf. on Dublin Core and Metadata Applications 2014, 2014.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 29

Deriving Model Metrics from Meta Models

Nebras Nassar[1], Thorsten Arendt[1], Gabriele Taentzer[1]

Abstract: The use of model-based software development has become more and more popular because it aims to increase the quality of software development. Therefore, the number and the size of model instances are cumulatively growing and software quality and quality assurance consequently lead back to the quality and quality assurance of the involved models. For model quality assurance, several quality aspects can be checked by the use of dedicated metrics. However, when using a domain specific modeling language, the manual creation of metrics for each specific domain is a repetitive and tedious process. In this paper, we present an approach to derive basic model metrics for any given modeling language by defining metric patterns typed by the corresponding meta-meta model. We discuss several concrete patterns and present an Eclipse-based tool which automates the process of basic model metrics derivation, generation, and calculation.

Keywords: Model metrics, metric patterns, quality assurance, Eclipse Modeling Framework.

1 Introduction

The paradigm of model-based software development (MBSD) has become more and more popular since it promises an increase in the efficiency and quality of software development. In this paradigm, models play an increasingly important role and become primary artifacts in the software development process. In particular, this is true for model-driven software development (MDD) [SVC06] where models are used directly for automatic code and test generation, respectively. Furthermore, the use of domain specific modeling languages (DSMLs) [BCW12] is a promising trend in modern software development processes to overcome the drawbacks concerned with the universality and the broad scope of general-purpose languages like the Unified Modeling Language (UML) [UML].

Since software models play an increasingly important role, software quality and quality assurance consequently lead back to the quality and quality assurance of the involved models. In [Ar11, Ar14], we introduced a structured syntax-oriented process for quality assurance of software models that can be adapted to project-specific and domain-specific needs according to a dedicated quality model (QM). The approach concentrates on quality aspects to be checked on the abstract model syntax and is based on quality assurance techniques model metrics, smells, and refactorings well-known from literature.

Metrics are useful to obtain quantitative information about software development processes and artifacts. Metrics for measuring the success of modeling and analysis has always been a challenge, especially in the area of enterprise modeling where very large models are in practical use. They can be used to analyze model quality, especially to find anomalies in

[1] Philipps-Universität Marburg, {nassarn,arendt,taentzer}@informatik.uni-marburg.de

models, and to estimate project costs. To measure quality aspects (e.g., model complexity) of DSMLs, basic model metrics are needed such as: total number of model elements of a specific type, number and average number of model elements of a specific type owned by a model element, number of incoming (outgoing) links of a specific type to (from) a model element, and average number of incoming (outgoing) links of a specific type per model element within the entire model. More really domain-specific metrics can be composed from already defined ones.

For evaluating quality issues we adopt the Goal-Question-Metrics approach (GQM) that is widely used and has been well established in practice [Va02]. Figure 1 illustrates the steps of the GQM process described as follows: 1. To measure the quality of a model, the first step is to define a measurement goal such as *model comprehensibility* [MDN09]. 2. Questions should be defined to support data interpretation towards a measurement goal. For example, the following question could be derived since a complex model is hard to understand: *How complex is a model wrt. the number of its elements?* 3. Metrics should be defined that help to provide all the quantitative information to answer the questions in a satisfactory way. A way to measure complexity is to use the metric *cyclomatic complexity* defined as *number of links* $-$ *number of elements* $+$ *2* [Mc76, Mc82] wrt. a control flow graph. To define model metrics for a given domain, the following tasks should be done: (a) find out (basic and complex) metrics needed to collect the related quantitative information and define them, (b) find and derive the corresponding domain metrics wrt. the definitions by analyzing and understanding the given domain structure, (c) identify the specification of each derived metric, and (d) find out which domain information is needed for specifying and implementing the derived metrics. 4. Once this information is identified, the corresponding code and artifacts of metrics calculation have to be developed for each derived metrics. 5. Implemented metrics can be used to analyze specific models. 6. After the defined metrics have been measured, sufficient information should be available to answer the questions. A cyclomatic complexity of 10, for example, points to a pretty complex control flow which might be hard to understand.

Fig. 1: GQM process

Considering a DSML being either a completely new language or a changed one due to evolution steps [HW14], its language and tool designers have to offer enough tool support for convenient domain-specific modeling processes. Specifying and defining metrics for a DSML by hand is time-consuming and error-prone. Although the definition and implementation of metrics cannot be automated completely, new ways are interesting to reduce the manual effort as much as possible. An approach and corresponding tool support for automatically deriving basic metrics from any given modeling language definition seems to be promising. Automatically deriving basic metrics, the effort of specifying and implementing model metrics would be reduced to composing basic metrics in a suitable way. Considering again the GQM process in Figure 1, Steps 1, 2 and 6 would remain manually, while 3 - 5 would be largely tool-supported.

The contributions of this paper are the followings:

- An approach to derive basic model metrics for any given modeling language (DSML or GPML [2]) being defined by a meta-model. The approach is especially useful for DSMLs to support the development of useful model metrics.
- An Eclipse-based tool to automate the definition of basic metrics for any given domain. The outcome of the tool is a high-level tool specification as an Eclipse plug-in which comprises the specification of the derived metrics, and code generation for metrics calculation, reporting and composition. (Steps 3 - 5 in Figure 1). The generated plug-in is used as input to EMF Refactor [Ref] for metrics calculation and composition.

Generating basic domain-specific metrics from a given meta-model helps us to concentrate on those metrics that really demand domain-specific knowledge. Our generation approach does not stop at trivial metrics but can also incorporate more complex ones such as the cyclomatic complexity and LCOM (Lack of cohesion of methods), or the definition of model queries. We illustrate our approach by using a DSML for simple Petri nets and discuss selected metrics patterns which are useful to derive basic metrics for this domain. Moreover, we present an implementation of the approach based on the Eclipse Modeling Framework (EMF) [EMF, St08] and EMF Refactor. To demonstrate the applicability and usefulness of the approach, we present an example application of this implementation on the UML class model domain.

The paper is structured as follows: The following section presents an example modeling scenario motivating our work. In Section 3, we present our approach for deriving basic model metrics in detail. Selected metrics patterns and their application on the example scenario are discussed in Section 4. After presenting the Eclipse-based tool prototype in Section 5, we demonstrate the applicability and usefulness of the approach in Section 6. Section 7 compares to related work and Section 8 concludes the paper.

2 Running Example: Basic Metrics for Petri Nets

In this section, we motivate our work by using an example Petri net scenario. We first describe the corresponding Petri net meta model. Thereafter, we discuss several basic metrics which can be used to analyze concrete Petri nets, i.e., instance models of this meta model.

Figure 2 shows the meta model of a Petri net language. A Petri net is composed of several places and transitions. Arcs between places and transitions are explicit: *PTArc* and *TPArc* are respectively representing place-to-transition arcs and transition-to-place ones. Arcs are annotated with weight. Each transition has at least one input place and one output place. Places can have an arbitrary number of incoming and outgoing arcs. In order to model dynamic aspects, places need to be marked with tokens. Figure 3 shows an example Petri net instance modeling some specific dynamic behavior. The Petri net consists of four

[2] Here, *GPML* refers to any general-purpose modelling language.

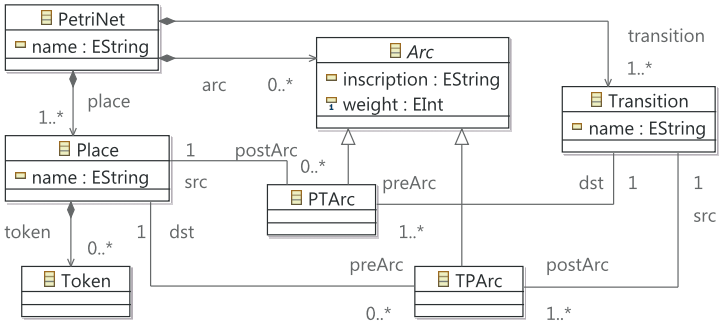


Fig. 2: Petri net meta model

places (P1 to P4), two transitions (T1 and T2) and altogether seven arcs connecting these elements. Please note that we omit arc weights and inscriptions for simplicity reasons.

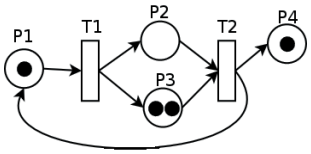


Fig. 3: Example Petri net instance

We present an example of the GQM process applied on the Petri net DSML as follows: Let the quality goal be the comprehensibility of Petri net models, one derived question is: *How complex is a model wrt. the number of its elements (places and transitions)?* A suitable metric to answer this question is the `cyclomatic complexity` defined as *number of tparcs and ptarcs − number of places and transitions + 2*. Applying this metric to the given Petri net instance (in Figure 3), the result is 3. Hence, there are only 3 independent paths and the given model is easy to understand and maintain.

To define metrics for analyzing models such as the *cyclomatic complexity*, some basic metrics are useful [Ch95, SJM92], e.g., metrics which simply count elements of specific types (like *number of transitions in the Petri net*, *number of places in the Petri net* and *number of arcs in the Petri net*). These basic metrics may be composed using arithmetic expressions to define the desired metrics.

When analyzing basic metrics such as the ones which calculate average values (like *average number of outgoing (or incoming) arcs of all transitions in the Petri net*), we observe that the structure of how they are specified is generic to some extent. The information needed can be obtained from three classes in the meta model (in our case *PetriNet*, *Transition*, and *TPArc*) where the first two classes are connected by a containment reference (*transition*) and the latter two classes are connected by a non-containment reference (*postArc*). In the following, we identify recurring patterns in meta models that are useful to derive basic model metrics.

3 Approach

Considering tool support for domain specific modeling, we are faced with the dilemma that we have to set up the tooling for metric calculation for each domain specific modeling language, even for basic metrics. And if the DSML has changed due to some evolution steps (e.g., the evolution of the Petri net meta model as described in [HW14]), its tooling has to be adapted. Therefore, we address the following research problem throughout this paper:

> How can the information, stored in a meta model, be used to automate the process of creating tool support for calculating basic metrics on corresponding instance models?

As we have seen in the preceding section, some basic metrics are defined for the Petri net domain such as *Number of transitions in the Petri net* and *Number of tokens in the Petri net*. These metrics could be abstractly described as *Number of instances of type X in an instance of type Y*. Moreover, we observe that several kinds of metrics can be derived by the same (or at least similar) abstract description. This abstract description can be used to specify basic metrics for any given domain by using the concrete domain data like the name of the corresponding domain element, e.g., *Transition*, *PetriNet*, and *Token*.

So, our approach is to design several metrics patterns (i.e., structural descriptions) which can represent the abstract description structure of several kinds of domain-specific metrics. Domains are usually defined by a domain-specific language, more specifically by a meta model. Therefore, the metrics patterns have to be typed by the meta-meta model so that the patterns can be applied over any given domain (meta-model) to find and derive the correspondences (the pattern matches). These correspondences within a specific domain hold the concrete domain data needed to define, specify and generate basic domain-specific metrics. These metrics are executable on instance models. In the following, we present the process for defining a new metrics pattern and deriving basic metrics from this pattern for a concrete meta model.

1. First, we design a pattern over the meta-meta model. This pattern consists, e.g., of two nodes and a containment relation in between as shown in Figure 4.

Fig. 4: A simple example of a metrics pattern

2. Then, the pattern can be applied to a concrete domain in order to find and retrieve all the pattern matches (correspondences) whose structures are instances of the pattern structure. For the Petri net example presented in Section 2, the following pattern matches are found:

   - Node *PetriNet*, Node *Place* and a containment relation named *place*.

- Node *PetriNet*, Node *Transition* and a containment relation named *transition*.
- Node *Place*, Node *Token* and a containment relation named *token*.

3. From each pattern match we now derive one or more basic model metrics. The following Petri net metrics can be derived from the matches described above:
   - Number of *places* in a selected *Petri net*.
   - Number of *transitions* in a selected *Petri net*.
   - Number of *tokens* in a selected *place*.

   Using the data of the pattern matches we can define and specify the model metrics for the given domain. Thereafter, an existing tool for metrics calculation on DSML instance models may be extended.

Our approach helps to easily produce the ”boilerplate” information of metrics specification. Having basic domain-specific metrics at hand, metrics and queries which require real domain-specific knowledge can be specified on top of those.

To sum up, metrics patterns are designed independent of concrete DSMLs. These patterns can be used to find and derive basic domain-specific metrics. Using the data of the retrieved pattern matches, we can derive basic domain-specific metrics and specify them in an appropriate query language like OCL. Furthermore, the corresponding code can be also generated in order to calculate metric values of concrete instance model. Figure 5 illustrates our approach.

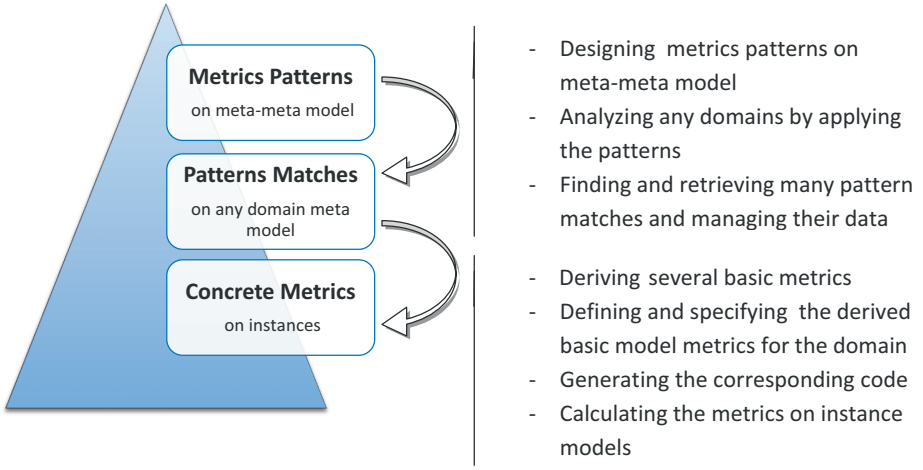


Fig. 5: The general approach to derive basic model metrics

Specifying thresholds to interpret metric results, however, is out of scope of this work due to individual interpretation opportunities depending on the modeling language, the modeling purpose, and the quality aspect considered by the measurement, respectively [Ar14].

4 Metrics Patterns

We developed 20 metrics patterns typed by Ecore, the meta-meta model of EMF [EMF, St08] representing the reference implementation of the Essential MOF (EMOF) standard for defining meta models using simple concepts [MOF]. The patterns are divided into four groups depending on the number of the *EReference* nodes in each pattern. In this section, we firstly present two selected patterns in detail: the simplest pattern and a more complex one. Finally, we give an overview about the remaining patterns.

4.1 Selected Metrics Patterns

Each metrics pattern can be used to describe the abstract structure of at least one kind of metrics. In summary, the 20 metrics patterns can derive altogether 42 kinds of basic model metrics. In the following descriptions of two selected patterns, the term *concrete pattern* refers to an instance of the basic metrics pattern.

Example for a simple metrics pattern

Figure 6 shows a simple concrete pattern (in concrete syntax) matched to the Petri net domain. It simply consists of one single class (*Place*) and can be used to derive and specify metric *Number of places in the model*. Now, our goal is to find all concrete patterns (matches) which have the same structure so that we can derive metrics that have the same abstract form. The pattern can be any class such as *Place* as shown in Figure 6.

Fig. 6: A node-related pattern

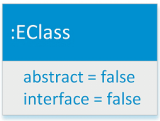


Fig. 7: Metrics pattern

Figure 7 shows the corresponding pattern (in abstract syntax) typed by Ecore. It consists of only one node of type *standard EClass* in order to represent non-abstract classes. [3] Applying this pattern to the Petri net domain, several concrete pattern matches exist as, e.g., *Transition*, *Token*, *PTArc* and *TPArc*. Each match can be used to specify a concrete metric. Example metrics for the Petri net domain are: number of *transitions* in the *model*, number of *tokens* in the *model*, number of *place-to-transition arcs* in the *model*, and number of *transition-to-place arcs* in the *model*.

As a result, this pattern can be used to find and derive domain metrics which have the following abstract description:

Number of all instances of type *X* in the model.

Here, *X* represents the name of any matched class from any given domain with respect to the applied pattern.

[3] Here, *standard EClass* means, that meta attributes *abstract* and *interface* are set to *false*.

Example for a more complex metrics pattern

Figure 8 shows a more elaborated concrete pattern (in concrete syntax) matched to the Petri net domain. It consists of class *PetriNet* connected by a containment relation named *transition* to class *Transition* which is in turn connected to class *TPArc* (the referenced class) by a non-containment relation named *postArc*.

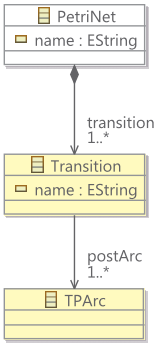


Fig. 8: An edge-related pattern

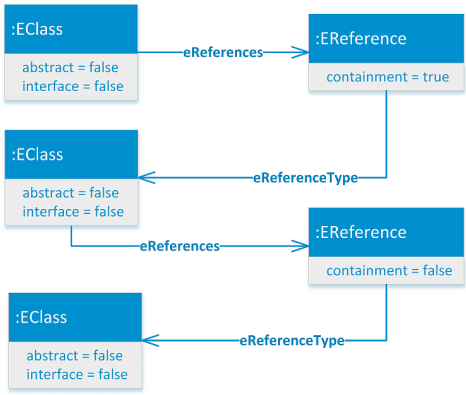


Fig. 9: Metrics pattern

This concrete pattern can be used to derive and specify the following concrete metrics:

- Number of all *outgoing arcs* of all *transitions* in a selected *Petri net*.
- Average number of *outgoing arcs* of all *transitions* in a selected *Petri net*.

Again, we want to find all concrete pattern matches for a given domain. Any concrete pattern must consist of three classes with one containment relation and one non-containment relation between these classes. Figure 9 shows the corresponding pattern (in abstract syntax), again typed by Ecore. The pattern consists of altogether five nodes. The nodes on the left are of type *standard EClass* (see above). Here, the top-left node is used to infer a container class, the middle-left node is used to infer a contained class , and the bottom-left node is used to infer a referenced class. The other nodes are of type *EReference*. Here, the containment attribute of the top-right node is set to *true* whereas the containment attribute of the other one is set to *false*. This means that the top-right node is used to infer the containment relation between the corresponding matched classes whereas the bottom-right node is used to infer the non-containment one.
As a result, this pattern can be used to find and derive metrics which have the following two abstract descriptions:

Number of all instances of type *X* referenced by all instances of type *Y* in a selected instance of type *Z*.
Average number of all instances of type *X* which are referenced by all instances of type *Y* in a selected instance of type *Z*.

Here, *X* represents the name of the matched referenced class, *Y* is the name of the matched contained class and *Z* is the name of the matched container class.

4.2 Metrics Pattern Groups

As mentioned above, we developed 20 metrics patterns which are separated into four groups depending on the number of *EReference* nodes in each pattern. In the following, we present an overview about the patterns in these groups and the kind of metrics which can be derived.

Group "Single node pattern" The first group contains only one pattern which consists of only one node of the type *EClass* to derive the atomic metrics (see Figure 7).

Group "One-edge patterns" The second group consists of seven patterns where each one contains one *EReference* node and several nodes of type *EClass*. These patterns can be used to derive the following kinds of metrics: average, percentage, sum or total number of instances of a specific type in or for a selected instance. Please note that some patterns are designed to support inheritance in order to match child classes of classes, abstract classes or interfaces. For example, some patterns can derive the following kind of metrics: *number of instances of a child type in a selected instance*. Concrete examples are: *number of transition-place arcs in a selected Petri net*, *number of place-transition arcs in a selected Petri net*, and *total number of all arcs in a selected Petri net* (the sum of both metrics mentioned before).

Group "Two-edge patterns" The third group also consists of seven patterns. Here, each pattern contains two *EReference* nodes and several nodes of type *EClass*. These patterns can be used to derive the same kinds of metrics provided by group 2 as well as more complex ones. For example, a pattern is designed using two *EReference* nodes which have the same source *standard EClass* and the same target *standard EClass* node. The containment attribute of one *EReference* node is set to true whereas the containment attribute of the other one is set to false. This pattern can derive the following kind of metrics: *Number of "part"-instances in a selected "whole"-instance so that they have the same specific role specified by non-containment reference*. Some patterns of this group are designed to match child classes of different pattern nodes such as the child classes of the whole part node, the direct-part node or of both. The pattern in Figure 9 belongs to this group.

Further more-edge patterns can be defined in the similar way. We assume that our current patterns may be matched often over any given domain because their structures are vital and cardinally needed for representing several parts of any meta-model structure. Additionally, some metrics derived by more-edge patterns could be defined by composing metrics derived by less-edge patterns.

Group "Composed patterns" The last group consists of five patterns for deriving more complex metrics respectively more specific ones. It contains some patterns for deriving metrics used to calculate the sum (average, percentage) of the number of different kinds of instances having the same whole instance as for example *total number of transitions and places in a selected Petri net*. The patterns can also be used to derive metrics like *average number of places with respect to the number of transitions in a selected Petri net*. However, these kinds of metrics may not make sense for each domain.

So far, we defined 20 metrics patterns to derive 42 different kinds of metrics. However, we do not claim that this list is complete. Furthermore, the existing patterns could be used to produce further kinds of metrics. The 42 kinds of metrics are only some suggested ones. We can also think of combining several kinds of metrics. Additionally, we can also use the approach to design further metrics patterns by using, as an example, a different number of nodes and relations with different attributes values. Additionally, we can design patterns for producing metrics used to inquiry on attributes values of nodes.

5 Tooling

In this section, we present the Nas tool that we developed to automate the process of model basic metrics creation, i.e., metrics derivation and specification for any given domain, and to automatically generate a high-level tool specification as an Eclipse plugin thereafter. The entire tooling is based on the Eclipse Modeling Framework (EMF) [EMF, St08].

Nas Tool We developed an Eclipse-based tool, called *Nas Tool*, which uses metrics patterns to automatically find matches and to automatically derive, specify and generate basic metrics of any given domain modeled in Ecore thereafter. The metrics patterns are designed by Henshin [Ar10, Hen], a model transformation engine for EMF based on graph transformation concepts, as pattern-based rules. Each rule mimics the EMF abstract syntax of a structure to be matched in a given meta-model. During the pattern matching process, an isomorphic mapping from EMF node and edge symbols in the pattern to actual nodes and edges in the meta-model is computed. The Nas tool can be easily applied: The only input is a meta model in Ecore. The outcome of the Nas tool is a high-level tool specification as an Eclipse plug-in which comprises the following: A specification tool support for model metrics containing already a number of basic metrics specified by OCL being derived from the meta model. This tool support can be used to define further model metrics as compositions of existing ones. An application tool support for model metrics which can be used to calculate the defined metrics on concrete domain-specific models. The generated plug-in is used as input to EMF Refactor which is an Eclipse tool that supports metrics calculation, reporting and composition. Figure 10 depicts the use of the Nas tool in combination with EMF Refactor.

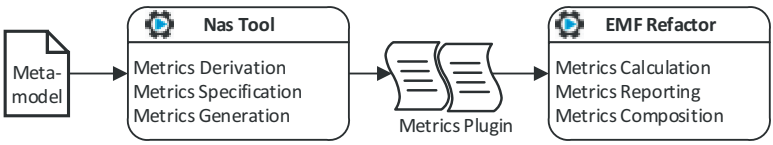


Fig. 10: The use of the Nas tool in combination with EMF Refactor

In addition, the Nas tool provides a view component for statistical information about derived metrics: It shows the number of pattern matches, the number of derived metrics for each applied pattern and the total number of matches and metrics.

Consequently, the Nas tool provides the following features:

- *Automation*: Creating (i.e. deriving, specifying and generating) basic model metrics automatically. Thus, the design and implementation time for each generated metrics is reduced.
- *Abstraction*: The Nas tool can be used to define and implement basic model metrics for any domain-specific language.
- *Simplicity:* The tool is easy to use, i.e., the user does have not to know about the metrics definitions, meta model structure and query languages.
- *Extendability:* The tool is extensible, i.e., it provides the ability to add further metrics patterns.

More information about the Nas tool, especially about its installation and the provided patterns, can be found at the accompanying web site of this paper [Nas].

6 Application Case

In this section, we demonstrate metrics generated by the Nas tool and compare them to the metrics provided by EMF Refactor [Ref] for a simple class modeling language (SCM) [AT13]. SCM represents the class diagram part of UML but it is much more simpler since it omits concepts like operations and association classes. Figure 11 shows the SCM meta model specified in Ecore.

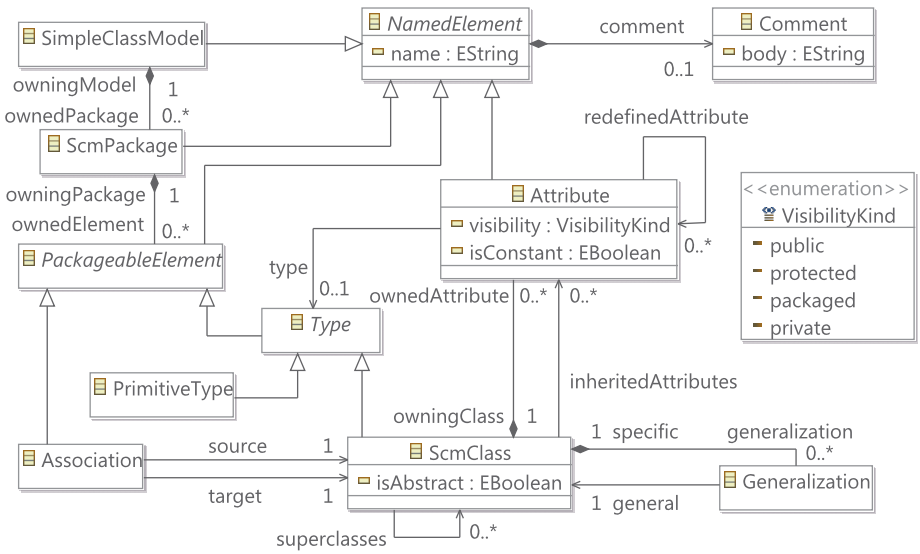


Fig. 11: The SCM meta model

The SCM metrics provided by EMF Refactor are manually specified and implemented using different perspectives and technologies like *Henshin*, *OCL* and *Java*. However, when

running the Nas tool on the SCM meta model, 127 basic metrics are automatically generated using only a few mouse clicks. 14 patterns are matched over the SCM model and some of them can generate several kinds of metrics. The 127 metrics are derived by altogether 72 matches of these metrics patterns. The number of pattern matches will be different from one domain to another one but we are convinced that the first six patterns (see [Nas]) have a high possibility to be matched over most given domains because the structure of each pattern of them could be considered as a simple basic structure.

After analyzing the SCM metrics provided by EMF Refactor, we found out that those metrics can be abstractly represented by 9 different metrics patterns. 6 of them are common with the patterns of the Nas tool whereas the other three patterns are too specific to the SCM model. The total number of the created metrics from these common patterns is 12 in EMF Refactor and 52 from the Nas tool.

Table 1 presents the SCM metrics provided by EMF Refactor and shows whether the metrics can be directly generated by the current version of the Nas tool, whether it can be defined by combining two generated metrics, or whether it cannot be generated. 12 out of 19 SCM metrics provided by EMF Refactor are also generated by the Nas tool. In addition, three metrics (i.e., 14, 15 and 17) are combinations of generated basic metrics. Hence, they profit from the Nas tool. We only need to combine two generated metrics using a mathematical operation, namely the division operation, like the metric called *Average number of attributes in concrete classes*.

In the following, we discuss the uncovered metrics: Metrics 2 and 18 could be derived by adding further metrics patterns, e.g., metric 2 can be derived by adding a pattern which takes incoming references into account. This metrics pattern shall be included into a future version of the Nas tool. Metrics 8 and 16 are too specific to the SCM model in the sense that they check a specific attributes value of the SCM model, e.g., if attribute *isAbstract* of *ScmClass* is set. In the future, we intend to consider ratio metrics for attribute values that are boolean or enumerations with literals.

The SCM metrics provided by EMF Refactor do not contain all the basic metrics of all SCM elements like the derived metrics from pattern 1, e.g., *number of all comments in the model*. The metrics in EMF Refactor are created using several different technologies and the process of creation took its time whereas creating the 127 SCM metrics generated by the Nas tool requires only a few mouse clicks. Please note that the non-covered metrics can be generated by adding new patterns or by extending some existing ones in the Nas tool. To summarize, the most of our selected patterns are matched with the SCM meta model and more than 78.5% of the metrics provided so far can be generated by the Nas tool or composed from generated ones.

Considering SDMetrics [SDM], a tool dedicated to the calculation of UML metrics, 96 metrics are provided by that tool, all specified by hand. We show that over 80% of them can be generated by the current version of the Nas tool or composed from generated ones. The rest of metrics could also be derived if the existence patterns were extended or new metrics patterns were added. More information about this comparison can be found at the web site of this paper [Nas].

Id	Metrics description	State
1	Number of direct children	covered
2	Number of external attributes with class type	not covered
3	Number of outgoing associations	covered
4	Number of incoming associations	covered
5	Number of associated classes	covered
6	Number of redefining attributes	covered
7	Total number of classes in the package	covered
8	Number of abstract classes in the package	not covered
9	Number of concrete classes in the package	covered
10	Number of associations in the package	covered
11	Total number of attributes in concrete classes	covered
12	Number of owned attributes in concrete classes	covered
13	Number of inherited attributes in concrete classes	covered
14	Average number of attributes in concrete classes	combination
15	Average number of associations per concrete class	combination
16	Ratio of the number of abstract classes	not covered
17	Ratio of the number of inherited attributes	combination
18	Number of equal attributes with other classes	not covered
19	Total number of model elements	covered

Tab. 1: The SCM metrics provided by EMF Refactor [AT13, Ref]

7 Related Work

To the best of our knowledge there is no directly related work on deriving model metrics from the corresponding meta model specification of a DSML. However, in this section, we discuss several topics being related to our work to some extent.

Model metrics. The problem of measuring the quality of models has been approached in several ways. Most of the presented metrics measure the quality of UML models. A survey of metrics applicable to UML models can be found in [GPC05]. Furthermore, in [La06, MP07], some general observations on managing quality, defining and reusing metrics for UML models are drawn.

The existing tool support for model metrics calculation is mainly aiming at UML and EMF modeling. Considering UML modeling, metrics calculation tools are integrated in standard UML CASE tools such as the IBM Rational Software Architect [RSA] and MagicDraw [MD]. A tool for UML metrics calculation is SDMetrics [SDM] (see above).

Related work in the context of quality metrics for meta models can be categorized into two groups: Firstly, work that deals with quality on the meta level, i.e., on meta models. Secondly, approaches that address quality on the instance level, i.e., on models. In [MA07], quality dimensions for MDD are derived. In [BV10], a taxonomy of meta model quality

attributes is presented. Both works use metrics to analyze the quality of the meta model instead of corresponding instance models. In [Vé06], a repository for meta models, models, and transformations is presented. The authors transfer metrics that were designed for class diagrams to meta models and apply them to contents of the repository. However, these works do not address the opposite direction to use structural information that is implicitly given in the meta model to derive basic metrics for instance models.

Metric definition and generation approaches. The closest relation to our work is presented in [YA14]. Here, the authors present a quality measurement framework for defining quality metrics at the meta model to measure the quality of conforming instance models. However, the motivation of this work is not to derive basic quality metrics like in our approach, but for evaluating a model by comparing it with a reference model which is motivated in the context of empirical studies, for example. In [AST10, Ar11, AT13] we present an EMF Metrics plug-in, as a part of EMF Refactor, supporting specification and calculation of model metrics for a given meta-model. Here, the derivation and specification of each model metric for each given meta-model have to be done manually by users. That work does not consider the automatic creation of model metrics for DSMLs.

In [Mo11], the authors present a model-driven measurement approach allowing modelers to dynamically add measurement capabilities to a DSML. The core of this work is to develop a metric specification meta model which enables to declare metric specifications as instance models. A metric specification model is taken as input to a prototype generator which outputs a full-fledge measurement software integrated into Eclipse. In this approach, the user should manually declare the metrics as instance models for each given meta model, whereas in our approach, the declaration of metrics will be automatically derived and specified from any given meta model.

In [Al09], the authors produce source-code representations of object-oriented applications. The generated representations should conform to a meta model that represents object-oriented languages such as Java and C++. A metric declarative language is developed to add new metrics without modifying the code of the framework. The metrics are executed on the generated representations. In that approach, the metric descriptions are declared concerning the object-oriented meta model, whereas in our approach, we derive metrics from any given meta model using characteristic patterns typed by Ecore (a meta-meta model) and using metrics descriptions.

8 Conclusion

In this paper we presented an approach for deriving basic metrics from the meta-model of a given domain-specific modeling language. In our work, 20 patterns are designed and described to derive metrics from any domain (meta-model) based on Ecore. The patterns are typed by the meta-meta model of EMF. Each pattern can be used to produce one or several kinds of basic model metrics.

We developed the Nas tool which takes the meta-model of any domain-specific language to automatically derive, specify and generate basic metrics based on the Eclipse technol-

ogy [Nas]. The use of the Nas tool is quite simple, only a few mouse clicks are required to create the metrics. Furthermore, the tooling provides an extension mechanism to add new custom metrics patterns. In this context, future work will concentrate on extending the existing metrics pattern base, supporting other metric kinds and more facilities for default calculation and compositions of metrics. Furthermore, we intend to figure out a vital set of metrics to be generated by analyzing, for example, user requirement specifications for the modeling language, to derive thresholds for the derived metrics, and to conduct further case studies.

References

[Al09] Alikacem, E-H; Sahraoui, Houari et al.: A Metric Extraction Framework Based on a High-Level Description Language. In: Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on. IEEE, pp. 159–167, 2009.

[Ar10] Arendt, Thorsten; Biermann, Enrico; Jurack, Stefan; Krause, Christian; Taentzer, Gabriele: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Model Driven Engineering Languages and Systems, pp. 121–135. Springer, 2010.

[Ar11] Arendt, Thorsten; Kranz, Sieglinde; Mantz, Florian; Regnat, Nikolaus; Taentzer, Gabriele: Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support. Software Engineering, 183:63–74, 2011.

[Ar14] Arendt, Thorsten: Quality Assurance of Software Models-A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project. PhD thesis, Philipps-Universität Marburg, 2014.

[AST10] Arendt, Thorsten; Stepien, Pawel; Taentzer, Gabriele: EMF Metrics: Specification and Calculation of Model Metrics within the Eclipse Modeling Framework. In: of the BENEVOL workshop. 2010.

[AT13] Arendt, Thorsten; Taentzer, Gabriele: A tool environment for quality assurance based on the Eclipse Modeling Framework. Automated Software Engineering, 20(2):141–184, 2013.

[BCW12] Brambilla, Marco; Cabot, Jordi; Wimmer, Manuel: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, 1(1):1–182, 2012.

[BV10] Bertoa, Manuel; Vallecillo, Antonio: Quality Attributes for Software Metamodels. Málaga, Spain, 2010.

[Ch95] Chamillard, Albert Timothy: An Exploratory Study of Program Metrics as Predictors of Reachability Analysis Performance. Springer, 1995.

[EMF] Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf.

[GPC05] Genero, Marcela; Piattini, Mario; Calero, Coral: A Survey of Metrics for UML Class Diagrams. Journal of Object Technology, 4(9):59–92, 2005.

[Hen] Henshin. http://www.eclipse.org/henshin.

[HW14] Herrmannsdörfer, Markus; Wachsmuth, Guido: Coupled Evolution of Software Metamodels and Models. In: Evolving Software Systems, pp. 33–63. Springer, 2014.

[La06] Lange, Christian FJ: Improving the Quality of UML Models in Practice. In: Proceedings of the 28th international conference on Software engineering. ACM, pp. 993–996, 2006.

[MA07] Mohagheghi, Parastoo; Aagedal, Jan: Evaluating Quality in Model-Driven Engineering. In: Modeling in Software Engineering, 2007. MISE'07: ICSE Workshop 2007. International Workshop on. IEEE, pp. 6–6, 2007.

[Mc76] McCabe, Thomas J: A Complexity Measure. Software Engineering, IEEE Transactions on, (4):308–320, 1976.

[Mc82] McCabe, Thomas J: Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. US Department of Commerce, National Bureau of Standards, 1982.

[MD] MagicDraw. http://www.nomagic.com/products/magicdraw.html.

[MDN09] Mohagheghi, Parastoo; Dehlen, Vegard; Neple, Tor: Definitions and approaches to model quality in model-based software development - A review of literature. Information and Software Technology, 51(12):1646–1669, 2009.

[Mo11] Monperrus, Martin; Jézéquel, Jean-Marc; Baudry, Benoit; Champeau, Joël; Hoeltzener, Brigitte: Model-driven generative development of measurement software. Software & Systems Modeling, 10(4):537–552, 2011.

[MOF] Meta Object Facility (MOF). http://www.omg.org/mof.

[MP07] McQuillan, Jacqueline A; Power, James F: On the Application of Software Metrics to UML Models. In: Models in Software Engineering, pp. 217–226. Springer, 2007.

[Nas] Nas Tool. http://www.uni-marburg.de/fb12/swt/nassarn/nastool.html.

[Ref] EMF Refactor. http://www.eclipse.org/emf-refactor.

[RSA] Rational Software Architect (RSA). http://www-03.ibm.com/software/products/us/en/ratisoftarch.

[SDM] SDMetrics. http://www.sdmetrics.com.

[SJM92] Soo, Lee Gang; Jung-Mo, Yoon: An empirical study on complexity metrics of Petri nets. Microelectronics Reliability, 32(3):323–329, 1992.

[St08] Steinberg, Dave; Budinsky, Frank; Merks, Ed; Paternostro, Marcelo: EMF: Eclipse Modeling Framework. Pearson Education, 2008.

[SVC06] Stahl, Thomas; Voelter, Markus; Czarnecki, Krzysztof: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, 2006.

[UML] Unified Modeling Language (UML). http://uml.org/.

[Va02] Van Solingen, Rini; Basili, Vic; Caldiera, Gianluigi; Rombach, H Dieter: Goal Question Metric (GQM) Approach. Encyclopedia of Software Engineering, 2002.

[Vé06] Vépa, Eric; Bézivin, Jean; Brunelière, Hugo; Jouault, Frédéric: Measuring Model Repositories. In: Proceedings of the 1st Workshop on Model Size Metrics (MSM'06) co-located with MoDELS. volume 2006, 2006.

[YA14] Yue, Tao; Ali, Shaukat: A MOF-Based Framework for Defining Metrics to Measure the Quality of Models. In: Modelling Foundations and Applications, pp. 213–229. Springer, 2014.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 45

Vom Clean Model zum Clean Code

Anna Vasileva[1], Doris Schmedding[2]

Abstract: In diesem Beitrag wird der Zusammenhang zwischen Code-Qualität und UML-Modellen in einem Software-Entwicklungsprozess in der Informatik-Ausbildung vorgestellt. Es wird untersucht, welche der im Code sichtbar werdenden Mängel bereits im Modell erkannt werden können. Werkzeuge zur statischen Code-Analyse und Refactoring-Techniken unterstützen die Studierenden beim Entdecken und Beseitigen der Qualitätsmängel im Programm-Code. Eine Analyse der studentischen Projekte hat gezeigt, dass sich manche Code-Mängel im Nachhinein nur schwer beseitigen lassen. Aus diesem Grund müssen Qualitätsaspekte bereits beim Modellieren in Betracht gezogen werden. Frühzeitig erkannte Mängel lassen sich mit geringeren Kosten beseitigen als spät erkannte Defekte.

Keywords: Clean Code, Code-Qualität, Qualität von UML-Modellen, statische Code-Analyse, Software-Entwicklung, Metriken

1 Einleitung

Das Software-Praktikum (SoPra) ist eine Lehrveranstaltung, in der im Team Software-Entwicklungsprojekte durchgeführt werden. Es findet in den ersten Semestern des Informatik-Studiums statt und wird nach einer Programmier- und Software-Technik-Veranstaltung besucht. Im Rahmen des SoPras werden die bis dahin erlernten Methoden, grundlegende Prinzipien und Software-Entwicklungsprozesse in der Praxis eingesetzt.

Es werden zwei Varianten des SoPras angeboten – als reguläre Veranstaltung im Wintersemester und als Blockveranstaltung in der vorlesungsfreien Zeit im Winter und im Sommer. Der offizielle Stundenumfang beträgt in beiden Fällen vier Semesterwochenstunden. Im Ferien-SoPra treffen sich die Studierenden jeden Tag und arbeiten etwa 30 Stunden pro Wochen an den Projekten.

Die Teilnehmer sind Bachelor-Studierende, die in Gruppen mit jeweils acht Mitgliedern aufgeteilt sind. Jede Gruppe hat die Unterstützung eines Betreuers bzw. einer Betreuerin. In der beschränkten Zeit werden zwei Projekte durchgeführt. Etwa sechs bis zehn Gruppen bearbeiten gleichzeitig dieselben Projekte. In der Regel ist das erste Projekt ein Verwaltungsprogramm und das zweite Projekt ein Spiel.

Die Software-Entwicklung basiert auf einer vereinfachten Version des

[1] Technische Universität Dortmund, Fakultät für Informatik, Otto-Hahn-Str. 12, 44227 Dortmund, anna.vasileva@tu-dortmundd.de
[2] Technische Universität Dortmund, Fakultät für Informatik, Otto-Hahn-Str. 12, 44227 Dortmund, doris.schmedding@tu-dortmund.de

Wasserfallmodells von Royce [Ro70]. Dieses ist auch für Studierende ohne Erfahrung leicht verständlich sowie für kleine und übersichtliche Projekte gut geeignet. Jede Phase ist vordefiniert und wird streng befolgt. Die Betreuer führen am Ende jeder Phase Reviews durch. Mit einer Präsentation der Zwischenergebnisse wird die aktuelle Phase jeweils abgeschlossen.

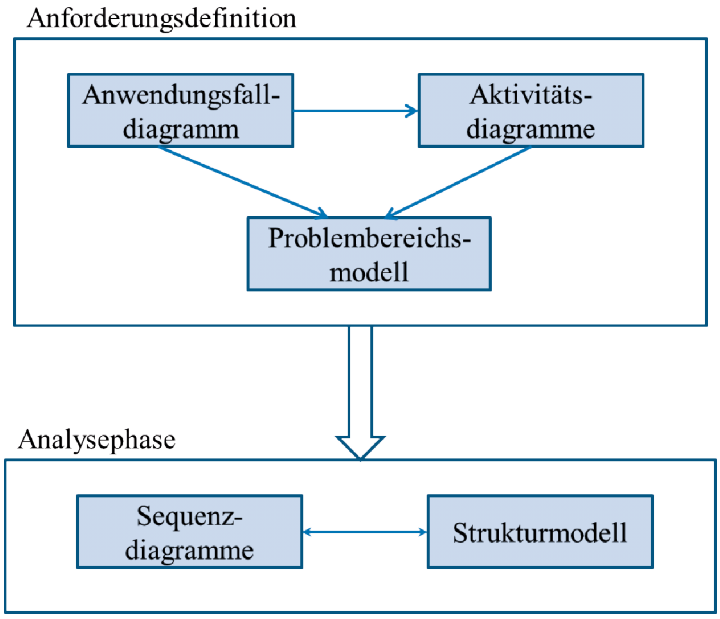


Abb. 1: UML-Modellierung

In der ersten Phase werden die Anforderungen definiert. Anschließend folgt die Analyse-Phase. In diesen beiden Phasen wird die Unified Model Language (UML) [Ru12] zur Modellierung eingesetzt. Abbildung 1 stellt die Abhängigkeit zwischen den in den einzelnen Phasen eingesetzten UML-Diagrammen dar. Darauf wird in Kapitel 3 näher eingegangen. Nach der Modellierung werden aus dem Strukturmodell einmalig Java-Code-Rahmen generiert. Das Programm wird implementiert und getestet. Dabei wird auch die Qualität des produzierten Codes überprüft. Am Ende der Implementierungsphase findet ein Produkttest durch das Entwicklerteam statt, in dem noch einmal überprüft wird, ob die in der Aufgabestellung geforderten Funktionalitäten von dem entwickelten Produkt erfüllt sind. Zum Abschluss des Projekts findet ein gegenseitiger Abnahmetest der teilnehmenden Gruppen statt.

Für die Implementierung wird die objektorientierte Programmiersprache Java eingesetzt. Die verwendete Entwicklungsumgebung ist Eclipse. Viele Plugins, z.B. für die Versions- und Zugriffskontrolle oder für die Entwicklung der grafischen Benutzeroberflächen, unterstützen die Studierenden zusätzlich.

Die Projekte haben einen Umfang von ca. 6000 Lines of Code (LOC) und umfassen ungefähr 30 Klassen ohne Berücksichtigung der Klassen der graphischen Benutzungsschnittstelle (GUI). Diese Klassen werden weitgehend mit Hilfe von GUI-Editoren erzeugt und bleiben deshalb in den nachfolgenden Betrachtungen unberücksichtigt.

Die Arbeit ist folgendermaßen aufgebaut. Nach der Einleitung gibt Kapitel zwei einen Überblick über die Motivation, die Ziele und die Probleme bei der Integration von Qualitätsaspekten in einen Software-Entwicklungsprozess. Im Kapitel „Modellierung mit UML“ wird genauer erläutert, welche UML-Diagramme von den Studierenden im Rahmen der ersten Phasen des Entwicklungsprozesses erstellt werden. In Kapitel vier und fünf wird vorgestellt, welche Maßnahmen bisher getroffen wurden, um die innere Qualität der erstellten Software zu verbessern. Anschließend folgt eine Erläuterung der Mängel, die bereits in der Modellierungsphase zu erkennen sind. Der Beitrag schließt mit einem Ausblick und einem Fazit.

2 Motivation

In den ersten Semestern des Informatik-Studiums lernen die Studierenden die Semantik der verwendeten Programmiersprachen, verschiedene Programmierparadigmen, die Modellierung mit UML, Entwurfsmuster und Software-Entwicklungsprozesse kennen. Im Zentrum der Ausbildung im Bereich Programmierung und Software-Entwicklung steht das Ziel, funktional korrekte Programme zu erstellen. Im Software-Praktikum ist bei der Abnahme der Projekte aber deutlich geworden, dass die innere Qualität der von den Studierenden erstellten Programme zu wünschen übrig lässt.

In mehreren Iterationen haben wir ein Konzept entwickelt, das der langfristigen Verankerung des Ziels „Hohe innere Software-Qualität“ im Software-Entwicklungsprozess dient. Unser Vorgehen basiert auf Untersuchungen zu typischen Qualitätsmängeln im Code von studentischen Projekten in der Lehrveranstaltung Software-Praktikum ([Sc15], [Re14]).

Die bisherigen Analysen der studentischen Projekte mit Hilfe eines Werkzeugs zur statischen Code-Analyse haben gezeigt, dass das Beseitigen von Mängeln, die bereits entstanden sind, häufig zeitaufwendig, sehr schwierig und für unerfahrene Entwickler manchmal auch unmöglich ist. Am Ende der Projekte bleibt keine Zeit für die Verbesserung der inneren Qualität der Programme. Der Schwerpunkt der Aktivitäten liegt in der Endphase des Projekts auf dem Aussehen, dem Hinzufügen von zusätzlichen Features und der funktionalen Korrektheit der vorhandenen Eigenschaften. Unser Ziel ist, dass die Studierenden bereits beim Modellieren mehr Wert auf die innere Qualität der Programme und nicht nur auf die funktionale Korrektheit legen.

3 Modellierung mit UML

In der ersten Phase des Software-Entwicklungsprozesses werden die Anforderungen erhoben. Ein Anwendungsfalldiagramm stellt die funktionalen Anforderungen in Form von Anwendungsfällen dar, die die Nutzer des Systems in unterschiedlichen Rollen ausführen können. Anschließend wird jeder Anwendungsfall näher beschrieben, indem der genaue Ablauf durch ein Aktivitätsdiagramm dargestellt wird.

Die Aktivitätsdiagramme sind wichtige Artefakte im Entwicklungsprozess. Beim Erstellen dieser Diagramme stellen die Studierenden zum ersten Mal fest, dass einige Anwendungsfälle sehr komplex sind. Andere Anwendungsfälle laufen anders ab, als die Studierenden gedacht haben. Zur genaueren Erläuterung der Aktivitätsdiagramme müssen die Studierenden eine textuelle Beschreibung der Anwendungsfälle erstellen, in denen Vor- und Nachbedingungen sowie Fehlerfälle beschrieben werden.

Der Entwicklungsprozess startet also mit einer ausführlichen Erhebung der funktionalen Anforderungen, auf die im weiteren Verlauf des Prozesses immer wieder zurückgegriffen wird. Neben der Modellierung der funktionalen Anforderungen wird ein Datenmodell, das so genannte Problembereichsmodell, erstellt, welches die Klassen des Problembereichs mit ihren Attributen und den Beziehungen untereinander darstellt.

In der Analysephase wird das Problembereichsmodell zum Strukturmodell ausgebaut, indem Steuerungsklassen ergänzt und Methoden hinzugefügt werden. Wir streben aus Gründen der Übersichtlichkeit und Verständlichkeit eine Dreischichtenarchitektur an, die sich an dem Model-View-Controller-Muster [Ga09] der Software-Entwicklung orientiert. Um zu überprüfen, ob im Strukturmodell alle Methoden und Beziehungen zwischen den Klassen vollständig erfasst sind, werden Sequenzdiagramme eingesetzt, die in der Regel die Anwendungsfälle aus dem Anwendungsfalldiagramm repräsentieren. Bei der Definition der Methoden im Strukturmodell werden die Signaturen der Methoden vollständig angegeben.

Im SoPra wird zum Modellieren mit UML das leichtgewichtige Werkzeug Astah [As15] verwendet. Dieses Tool ist sehr benutzerfreundlich und intuitiv bedienbar. Die Studierenden kommen damit gut zurecht. Das Strukturmodell und die Sequenzdiagramme sind, wie bereits erläutert, inhaltlich gekoppelt. Bei Astah liegt diesen beiden Diagrammtypen ein gemeinsames Datenmodell zugrunde. Bei der Erstellung eines Sequenzdiagramms stehen die bereits definierten Methoden zur Verfügung. Eine große Unterstützung stellt die Möglichkeit zur automatischen Generierung von Java-Code-Rahmen dar.

4 Code-Qualität

Die innere Qualität der Programme beruht an erster Stelle auf der Lesbarkeit und Verständlichkeit des Programm-Codes. Erst in der Zusammenarbeit im Team und wenn

die Zeit knapp ist, wird klar, dass eine hohe innere Qualität des Codes das Modifizieren und Testen vereinfacht und die Wartbarkeit des entstehenden Programms erhöht.

Mit Hilfe eines Werkzeugs zur statischen Code-Analyse können außerdem potentielle Fehler frühzeitig entdeckt werden.

Es wurden Messungen mit dem Tool zur statischen Code-Analyse PMD [PMD15] durchgeführt. Die erste Messung diente der Analyse des studentischen Programm-Codes und der Suche nach typischen Defekten. Beim Programmieren wussten die Studierenden anfangs nicht, dass Messungen durchführt werden und nach Mängeln gesucht wird.

Die Analyse der studentischen Projekte hat gezeigt, dass immer wieder ähnliche Mängel im Programmcode vorkommen. Diese lassen sich in folgende Bereiche aufteilen:

- Namensgebung und Einhaltung der Java-Konventionen,
- Komplexität und Länge der Methoden,
- Verantwortlichkeit, Länge und Komplexität der Klassen.

Die Anzahl der Metriken und der Tools, welche die innere Qualität vom Programm-Code messen, ist sehr groß. Gemäß der Goal-Question-Metrik-Methodik [Ba94] werden konkrete Ziele zur Verbesserung der Code-Qualität definiert. Die ausgewählten Metriken dienen dem Ziel, die Lesbarkeit, Verständlichkeit und Wartbarkeit des Codes zu erhöhen. Für unsere Lehrveranstaltung wurden Metriken und Grenzwerte ausgewählt, die für die Studierenden leicht verständlich sind [Re14]. Die Metriken helfen, die Mängel mit Tool-Unterstützung möglichst leicht zu entdecken und zu beheben. Tabelle 1 stellt die definierten Ziele, die verwendeten Metriken und Grenzwerte dar.

Bei der Auswahl der Metriken und der Festlegung der zu erreichenden Grenzwerte wurden der Projektumfang sowie die Erfahrungen der Studierenden berücksichtigt. Darauf aufbauend wurde für das ausgewählte Tool zur statischen Code-Analyse PMD ein SoPra-spezifischer Regelsatz definiert. Dieser wird im XML-Format im SoPra-Wiki [SP15] zur Verfügung gestellt.

Da ein Werkzeug zur statischen Code-Analyse nicht über die Aussagekraft eines Bezeichners entscheiden kann, müssen die Bezeichner auch manuell kontrolliert werden. Mit Hilfe von PMD kann nur die Länge der Bezeichner geprüft werden. Jedoch können auch Bezeichner, die länger als fünf Zeichen sind, sinnlos sein, die Java-Konventionen nicht einhalten oder missverständliche Information liefern.

Ziel	Metrik	Grenzwert	Erkannt durch
Einhaltung der Java-Konventionen	Naming Conventions	-	PMD + Manuelle Prüfung
Sinnvolle Bezeichner	Länge der Bezeichner	5 Zeichen	PMD + Manuelle Prüfung
Übersichtliche und gut lesbare Methoden	Zeilenlänge der Methoden	40 Zeilen	PMD
	Parameteranzahl der Methoden	4	PMD
	Zyklomatische Komplexität	10	PMD
Übersichtliche und gut lesbare Klassen	Zeilenlänge der Klassen	400 Zeilen	PMD
	Toter Code	-	PMD
	Gott-Klasse	WMC[3] > 47, ATFD[4] > 5, TCC[5] < 0,33	PMD

Tab. 1: SoPra-Regeln

5 Bisherige Vorgehensweise zur Qualitätsverbesserung

Eine erste Messung ohne Vorankündigung hat gezeigt, dass die ausgewählten Metriken [Re14] und Grenzwerte für die Studierenden grundsätzlich erreichbar und passend zum Projektumfang gewählt sind. Dennoch wiesen die Projekte erhebliche Mängel auf.

In mehreren Iterationen, Durchführungen des Software-Praktikums, wurde daran gearbeitet, das Thema Code-Qualität in den Ablauf des Software-Entwicklungsprozesses langfristig zu integrieren und die Qualitätsergebnisse der Gruppen zu verbessern [Sc15]. Die Iterationen orientieren sich an dem Plan-Do-Check-Act-Zyklus (siehe Abbildung 2). In jeder Iteration wurden mit Hilfe von PMD-Messungen durchgeführt. Diese wurden analysiert und anhand der Ergebnisse wurden Änderungen am didaktischen Vorgehen vorgenommen, die die Integration der Qualitätsaspekte zusätzlich unterstützen. Diese Änderungsmaßnahmen werden nachfolgend erläutert.

[3] Weighted Method Count
[4] Access To Foreign Data
[5] Tight Class Cohesion

Bereits in der Einführungsveranstaltung zum Praktikum wird Code-Qualität thematisiert. Zum Thema Clean Code [Ma09] und Refactoring [Fo99] werden Tutorials im SoPra [SP15] zum Selbststudium bereitgestellt.

Die Studierenden führen in der Implementierungsphase eigenständig Messungen mit PMD und dem zur Verfügung gestellten SoPra-Regelsatz durch. Die GruppenbetreuerInnen weisen wiederholt auf die Relevanz der Code-Qualität für das Projekt hin und versuchen gemeinsam mit ihrer Gruppe bessere Lösungen zu finden.

Trotz all dieser Maßnahmen wurde festgestellt, dass es offenbar nicht ausreicht, über die Möglichkeit von Qualitätsmessungen und Maßnahmen zur Qualitätsverbesserung informiert zu sein. Deswegen musste ein anderer Weg beschritten werden. Nach dem ersten und vor dem zweiten Projekt folgte eine Diskussion der Ergebnisse der durchgeführten Messungen mit jeder Gruppe und deren BetreuerIn. Die Studierenden wurden in der letzten Iteration aufgefordert, die gefundenen Mängel mit Hilfe von Refactoring-Techniken [Fo99] zu beheben und einen Bericht darüber zu verfassen. Wenn das Verwenden von Refactoring nicht erfolgreich war, mussten die Studierenden dies schriftlich begründen.

Ein Vergleich der Ergebnisse mehrerer Sopra-Durchläufe zeigt, dass diese Maßnahme endlich dazu geführt hat, dass die Ergebnisse im zweiten Projekt deutlich besser wurden. Besonders auffallend ist, dass es in der Kategorie Bezeichner fast keine Verstöße gegen die Bezeichnerwahl gab. Das liegt auch daran, dass die Studierenden diese Mängel bereits beim Modellieren erkennen konnten.

Viele Mängel, insbesondere im Bereich der Namensgebung, lassen sich problemlos, z.B. durch das Refactoring *Rename,* beheben. Um die Länge der Methoden zu verkürzen und ihre Komplexität zu verringern, kann das Refactoring *Extract Method* oft erfolgreich eingesetzt werden. In manchen Fällen waren die Studierenden trotz ihrer Bemühungen nicht in der Lage, die Mängel zu beseitigen. Jedoch konnten sie gut erklären, woran sie gescheitert waren.

Diese praktische eigene Erfahrung der Studierenden und der Einsatz dieser didaktischen Methoden haben dazu geführt, dass sich die Entwicklerteams bereits in der Modellierungsphase des zweiten Projektes Gedanken über eine gute Bezeichnerwahl und die Vermeidung von langen Methoden, langen Parameterlisten und Gott-Klassen gemacht haben. Bei der Implementierung haben einige Gruppen versucht, der Regel vom Martin [Ma09] zu folgen. Diese besagt, dass eine Methode nur vier Zeilen lang sein sollte.

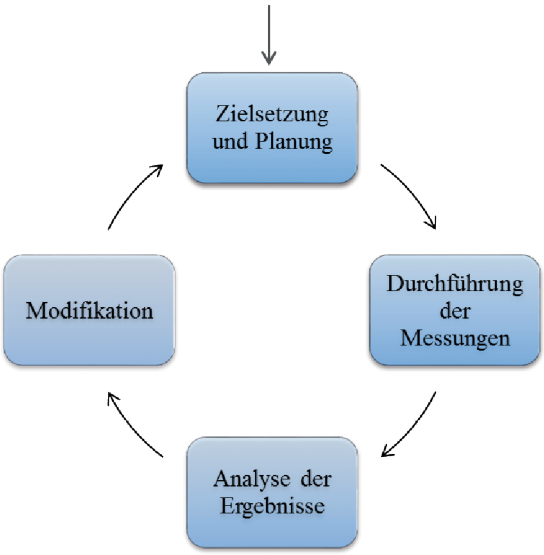


Abb. 2: PDCA-Zyklus

6 Entdeckung von Mängeln beim Modellieren

Frühzeitig erkannte Mängel sind einfacher zu beheben, als wenn diese bereits fest im Programm-Code verankert sind. Basierend auf unseren Untersuchungen [Sc15] wurden die Mängel analysiert, die bereits im Modell zu finden sind, und nach Hinweisen gesucht, die im Modell auf spätere mögliche Defekte im Programm-Code hindeuten. Diese Mängel werden in den folgenden Unterkapiteln vorgestellt.

6.1 Bezeichner

Die Wahl guter Bezeichner ist von großer Bedeutung für die Lesbarkeit des Programmcodes. Die Studierenden neigen dazu, kurze und nicht aussagekräftige Bezeichner zu wählen. Häufig wird auch Humor verwenden, der die allgemeine Verständlichkeit des Codes nicht unterstützt.

In Abbildung 3 wird eine Steuerungsklasse gezeigt, die im Rahmen eines Verwaltungsprojekts modelliert wurde. Die Studierenden musste ein Software-Produkt für die Organisation einer Cocktail-Bar erstellen. Mit Hilfe des Programms sollten Cocktail-Rezepte und ein Vorrat an Zutaten verwaltet werden. Die vorgestellte Klasse beinhaltet die Methode „getMaxMix“, die ein Beispiel für eine schlechte

Bezeichnerwahl darstellt. Der Name der Methode hält zwar die Java-Konventionen ein, ist aber inhaltlich nicht aussagekräftig. Sogar mit gutem Kontextwissen ist die Bedeutung schlecht zu verstehen.

In der ersten Iteration wurde festgestellt, dass die Java-Konventionen schon bei der Namensgebung in den UML-Diagrammen nicht eingehalten wurden. Da, wie bereits erwähnt, mit Hilfe von Astah aus dem Strukturmodell die Java-Code-Rahmen der Klassen und Methoden generiert werden, werden die schlecht gewählten Bezeichner aus dem Modell automatisch in den Java-Code übernommen.

Die Experimente haben gezeigt, dass die Qualitätskategorie Namensgebung sehr einfach und verständlich für die Studierenden ist. Mit Hilfe von Refactoring können ohne weitere Probleme und Nebenwirkungen die gefundenen Mängel schnell beseitigt werden. In der letzten Iteration haben die Studierenden nach der Einführung des Beseitigungszwangs bereits beim Modellieren auf die Bezeichnerwahl geachtet. Verstöße gegen die Java-Konventionen wurden deshalb im Programm-Code kaum noch gefunden.

Ein Grund für die hohe Qualität bei der Namensgebung für aus dem Modell übernommene Bezeichner in der letzten Iteration kann die Zusammenarbeit im Team beim Erstellen der UML-Modelle, aus denen die Java-Code-Rahmen generiert werden, sein. Hinzu kommen die von den Betreuern durchgeführten Reviews.

Besonders viele zu kurze Bezeichner stellten wir in Programmteilen fest, die von Einzelpersonen in der Implementierungsphase erstellt wurden. Diese Programmteile wurden bis dahin keinem definierten Code-Review-Prozess unterzogen.

6.2 Methoden

Als Qualitätsmerkmale der Methoden werden die Länge der Parameterlisten, die Länge der Methoden und ihre Komplexität betrachtet.

Mit Hilfe von Refactoring-Techniken, z.B. *Extract Method* [Fo99], kann die Lesbarkeit der Methoden verbessert werden, indem ihre Länge und Komplexität verringert werden. Diese Verbesserungen können aber nicht immer oder nicht so einfach umgesetzt werden. Ob nach dem Refactoring die Funktionalität erhalten geblieben ist, muss regelmäßig durch Tests überprüft werden. Die korrekte Beseitigung derartiger Mängel kostet viel Zeit.

Die Anzahl der Parameter lässt sich mit Hilfe von Refactoring reduzieren. Durch *Introduce Parameter Object* eine neue Klasse erzeugt und mehrere Parameter einer Methode lassen sich durch ein Objekt dieser Klasse ersetzen. Diese Methodik hat jedoch auch Nachteile. Die Lesbarkeit und die Übersichtlichkeit der Programme werden nicht unbedingt verbessert, da durch das Erzeugen von neuen Klassen die gesamte Struktur im Nachhinein verändert wird.

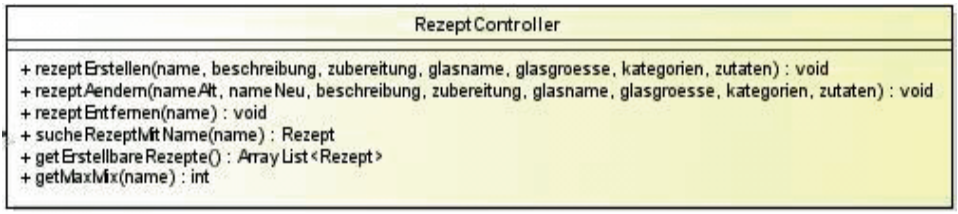


Abb. 3: Klasse in einem SoPra-Projekt

Die Länge der Parameterliste lässt sich sehr gut bereits im Modell überprüfen. Die vorgestellte Klasse in Abbildung 3 beinhaltet zwei Methoden mit sehr langen Parameterlisten. Laut unserer SoPra-Regeln (s. Tabelle 1) dürfen Methoden maximal vier Parameter übergeben bekommen. Eine Parameterliste darf nicht zu lang sein, weil sie schwer zu verstehen und zu benutzen ist. Insbesondere Boolesche Variablen, die wie Schalter funktionieren, sollten nach den Regeln von Robert Martin [Ma09] vermieden werden. Stattdessen sollten zwei Methoden angelegt werden, um die Komplexität zu verringern.

Die Wahrscheinlichkeit, dass Methoden mit sehr langen Parameterlisten auch zu umfangreich sind und eine hohe zyklomatische Komplexität besitzen, ist hoch. Die übergebenen Parameterwerte müssen überprüft werden. In unserem Beispiel muss beim Erstellen eines Rezepts überprüft werden, ob die Eingabe z.B. keine Sonderzeichen beinhaltet, kein leerer String übergeben wird oder ein Rezept mit dem Namen bereits hinzugefügt wurde. Diese Überprüfungen erhöhen die Komplexität und die Länge zusätzlich. Erfahrungsgemäß sind die Methoden, die mehr als 40 LOC haben, auch diejenigen, die von PMD als zu komplex erkannt werden.

Die Komplexität und die Länge der Methoden kann nicht erkannt werden, wenn nur das Klassendiagramm betrachtet wird, da ein Klassendiagramm nur die Struktur des Systems und nicht das Verhalten darstellt.

Zu Beginn eines Projekts werden in den Aktivitätsdiagrammen die Abläufe der Anwendungsfälle modelliert. Diese Anwendungsfälle entsprechen den Methoden in den Steuerungsklassen. Einen ersten Eindruck von der Komplexität eines Anwendungsfalls erhält man, wenn die Verzweigungen im Aktivitätsdiagramm betrachtet werden.

In der nächsten Phase, der Analysephase, erfolgt eine bereits implementierungsnähere Modellierung der Anwendungsfälle als Interaktion der Objekte in Sequenzdiagrammen. So wird das Verhalten der Methoden modelliert. Aus einem Sequenzdiagramm lässt sich zum einen ablesen, welche Teilaufgaben an andere Objekte delegiert werden, aber auch, welche Entscheidungen im Objekt selbst getroffen werden.

Beide Diagrammarten sind Teil der Verhaltensmodellierung und stellen Skizzen der Abläufe dar. Sie liefern nur Indizien auf zu hohe Komplexität. Anhand dieser Diagramme kann aber geschätzt werden, wie hoch ungefähr die zyklomatische Komplexität einer Methode sein könnte.

6.3 Klassen

Für die Studierenden war es besonders schwierig, die bereits entstandenen Mängel in dieser Kategorie durch Refactoring auf dem Programm-Code zu beseitigen.

Bei der Messung mit dem Tool zur statischen Code-Analyse wird geprüft, ob eine Klasse zu viel Verantwortung übernimmt, man spricht dann von einer Gott-Klasse. Die Definition von Gott-Klassen und die Kriterien, die die Entdeckung unterstützen, wurden von Lanza und Marinescu [LM06] übernommen.

PMD erkennt eine Gott-Klasse, wenn alle folgenden Kriterien (vgl. Tab. 1) verletzt sind:

- Die Summe der zyklomatischen Komplexität aller Methoden einer Klasse (WMC - Weighted Method Count) darf den Grenzwert von 47 nicht überschreiten.
- Die Anzahl der direkten Zugriffe einer Klasse auf die Attribute anderer Klassen (ATFD – Access To Foreign Data) darf nicht höher als 5 sein.
- Die dritte Metrik (TCC - Tight Class Cohesion) misst die Anzahl der direkt gekoppelten public-Methoden einer Klasse geteilt durch die maximale Anzahl der Verbindungen der Methoden. Dieser Wert sollte 0,33 nicht unterschreiten. Nur wenn dieser höher als 0,33 ist, wird nach der Definition von Lanza und Marinescu [LM06] die gewünschte Kohäsion der Methoden gewährleistet. Zwei Methoden sind als verbunden anzusehen, wenn sie auf die gleichen Instanzvariablen einer Klasse zugreifen. Innerhalb einer Klasse sollten die Methoden eine hohe Kohäsion besitzen. Andernfalls kann man die Methoden leicht auf zwei Klassen aufteilen.

Aus den durchgeführten Diskussionen mit den Studierenden und ihren Berichten ist bekannt, dass die Definition der Gott-Klassen für die Studierenden schwer zu verstehen ist. Einmal entstandene Gott-Klassen sind durch Refactoring schwer zu beseitigen. Die Berichte in der letzten Iteration zeigten, dass die Studierenden trotz offensichtlicher Anstrengungen damit überfordert waren, z.B. Gott-Klassen im Nachhinein zu beseitigen. Aus diesem Grund ist es notwendig, dass zu lange Klassen und zu komplexe Methoden bereits bei der Modellierung vermieden werden. Oft lässt sich schon bei der Modellierung erkennen, dass eine Klasse oder eine Methode zu viel Verantwortung übernimmt. Die Klasse in Abb. 3 ist ein Beispiel für eine Gott-Klasse, die bereits in der Modellierungsphase erkennbar ist.

Controller-Klassen, die für die Umsetzung der Anwendungsfälle verantwortlich sind, neigen häufig dazu, sich zu komplexen Klassen zu entwickeln. Die Modell-Klassen sind meist unproblematisch.

Besonders häufig haben wir Gott-Klassen in den Spiel-Programmen gefunden, wenn simulierte Gegner mit verschiedenen Spielstärken implementiert werden sollten. Oft ist es in diesen Klassen auch duplizierter Code zu finden, weil die verschiedenen Stärken ähnlich realisiert sind. Außerdem sind auskommentierte Methoden zu finden, die z.B. die schnellere Berechnung der möglichen Züge unterstützen sollten. „Duplizierter Code“

ist ein Mangel, der erst in der Implementierungsphase entsteht und im Modell meist noch nicht vorhergesehen werden kann. Sequenzdiagramme mit ähnlichen Abläufen könnten ein Indiz für diesen Mangel sein.

„Toter Code“ ist ein weiterer Mangel, auf den man bei den Klassen achten muss. Viele Methoden werden aus dem Strukturmodell automatisch generiert, von denen einige in der Implementierungsphase nicht benutzt werden. In Spielprogrammen ist es oft so, dass die Studierenden beim Modellieren des simulierten Gegners z.B. nicht wissen, welcher Algorithmus passend zu dem Spiel sein kann. Aus diesem Grund werden Methoden hinzugefügt, die sich später als überflüssig herausstellen. Derartige Mängel stören nicht beim Kompilieren, so dass sie nicht betrachtet und beseitigt werden, obwohl die Beseitigung einfach ist. „Toter Code“ ist ein Mangel, der erst in der Implementierungsphase entsteht und den es im Modell noch nicht gibt.

Wie bereits erwähnt, kann die zyklomatische Komplexität der Methoden nicht aus dem Klassendiagramm abgelesen werden. Dafür wird eine nähere Erläuterung der Methoden benötigt, welche die Aktivitäts- und Sequenzdiagramme liefern.

Die Aktivitäts- und die Sequenzdiagramme zeigen, wie kompliziert die Methoden sein können. Diese Erkenntnis muss beim Modellieren des Strukturmodells berücksichtigt werden, so dass nicht zu viele komplexe Methoden in einer Klasse zusammengefasst werden sollten. Erfahrungsgemäß ist, wenn zwei oder mehrere derartigen Methoden in einer Klasse zu finden sind, die Gefahr einer Gott-Klasse sehr groß (siehe Beispiel in Abb. 3).

Bei der Kontrolle der Code-Qualität am Ende des Projekts hat sich unsere Vermutung bestätigt. Für die Klasse „RezeptController“ hat PMD, abgesehen davon, dass die Anzahl der Methoden zu hoch war, für die Methode „rezeptErstellen“ eine zyklomatische Komplexität von 17 und für „rezeptAendern“ von 20 gemessen. Diese Methoden hatten auch ca. 65 LOC, was unseren Grenzwert von 40 LOC überschreitet. Außerdem war diese Klasse eine Gott-Klasse mit WMC 49, ATFD 18 und TCC 0.0.

Auch Klassen, die zu viele Attribute haben, können kritisch sein und sollten rechtzeitig betrachtet werden. Viele Attribute führen dazu, dass die Parameterlisten, z.B. im Konstruktor, lang werden. Eine derartige Klasse sollte in mehrere Klassen aufgeteilt werden.

Lange Klassen sind im SoPra eher selten, da der Projektumfang nicht so groß ist. Die längste Klasse, die wir gefunden haben, hatte 992 LOC. Diese war auch eine Gott-Klasse. WMC betrug 126, ATFD war 10-fach größer als der Grenzwert und TCC erreichte nur 0.042. In dieser Klasse wurden auch auskommentierter und duplizierter Code gefunden.

An den vorgestellten Beispielen ist erkennbar, dass die betrachteten Mängel voneinander abhängig sind. Eine Klasse wird zu lang, wenn sie entweder zu viele Methoden hat oder/und die Methoden zu lang sind. Wenn die Methoden zu lang sind, sind sie

erfahrungsgemäß auch zu komplex, dann kann sich die Klasse auch zu einer Gott-Klasse entwickeln. Die Komplexität der Methoden kann oft anhand der Anzahl der Parameter der Methoden, der Komplexität des zugehörigen Aktivitäts- und Sequenzdiagramms vorhergesehen werden (siehe Abbildung 4). Diese Abhängigkeit lässt sich von Anfang an kontrollieren, indem die Modell-Merkmale überprüft werden.

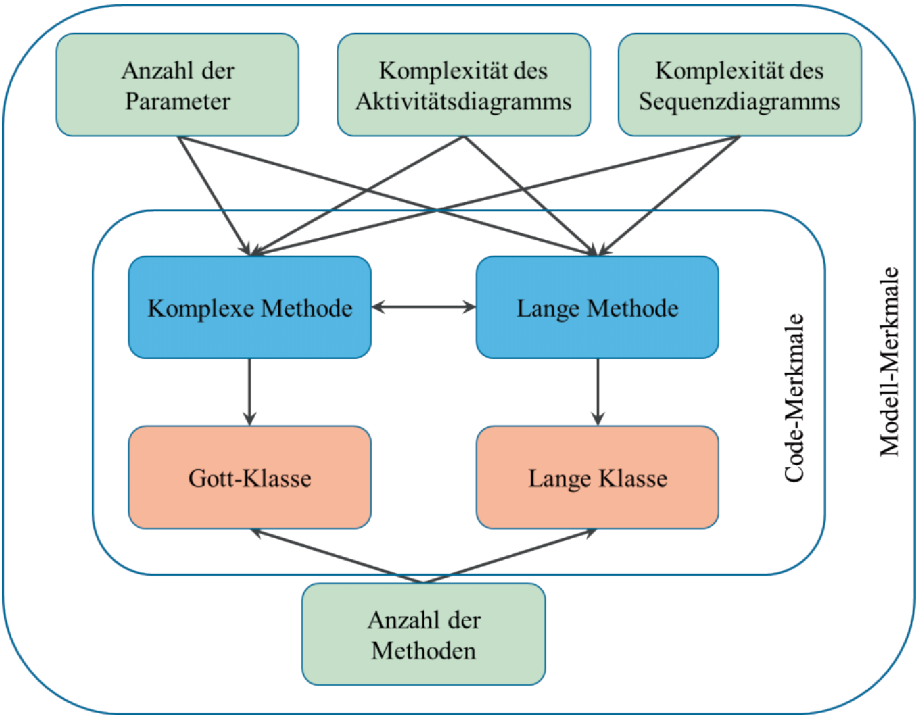


Abb. 4: Abhängigkeit zwischen den verwendeten Metriken

7 Ausblick

Wir wollen, dass die Studierenden lernen, die in den UML-Modellen abgebildete Komplexität richtig einzuschätzen, um von vornherein zu lange und zu komplexe Klassen zu vermeiden. Das ist das nächste Ziel, das wir uns für die Verbesserung unserer Lehrveranstaltung gesetzt haben.

Unsere bisherigen Arbeiten zur Code-Qualität ([Sc15], [Va15]) basieren auf den meist verwendeten Metriken im Bereich der objektorientierten Programmierung. Diese wurde von McCabe [Mc76] und Chidamber und Kemerer (CK-Metriken) [CK94] für Code-Qualität vorgestellt und analysiert. Die für unsere Lehrveranstaltung gewählten Grenzwerte und Bewertungskriterien haben sich als passend zum Umfang unserer

Projekte und der Erfahrungen der Studierenden erwiesen, so dass auch die Motivation nicht verletzt wurde. Für die Verankerung der Code-Qualität im Entwicklungsprozess hat sich der PDCA-Zyklus als erfolgreich erwiesen.

In der Modifikationsphase (s. Abbildung 2) haben wir in jeder Iteration die Reaktion der Studierenden berücksichtigt und passende Änderungen vorgenommen. In den zukünftigen Iterationen sollen Regeln, die auf den oben dargestellten Überlegungen zur gegenseitigen Beeinflussung der Qualitätsmetriken basieren und somit die Untersuchung der Qualität der UML-Modelle ermöglichen, verwendet werden.

Durch automatisches Analysieren der UML-Diagramme und der von Astah erstellten Code-Rahmen wollen wir die Studierende auf potentielle Fehler, die beim Modellieren übersehen wurden, aufmerksam machen.

Tang und Chen [TC02] stellen ein Werkzeug vor, das UML-Modelle ohne Berücksichtigung des Quellcodes überprüft. Das Tool basiert auf den CK-Metriken. Die betrachteten UML-Diagramme sind Klassen-, Aktivitäts- und Kommunikationsdiagramme und werden mit Hilfe des Tools Ration Rose erstellt. Tang und Chen nehmen an, dass alle Methoden in einer Klasse ähnliche Komplexität haben. Aus diesem Grund verwenden sie zu ihren Messungen die Metrik WMC1 statt WMC für alle Methoden im Modell. Nach der Definition von WMC1 hat jede Methode eine Komplexität von 1, d.h. es wird für die Komplexität der Klassen die Anzahl der Methoden pro Klasse gemessen. Das scheint erfahrungsgemäß ein sehr ungenauer und ungeeigneter Schätzwert zu sein. Wir streben die Entwicklung einer besseren Metrik für die Komplexität der Methoden im Modell auf Basis der zur Verfügung stehenden Aktivitäts- und Sequenzdiagramme an.

8 Fazit

In diesem Beitrag wurde eine langfristige Integration von Qualitätsaspekten für Programm-Code durch didaktische Maßnahmen vorgestellt. Am Ende der Implementierungsphase lassen sich die Mängel im Programm-Code mit Hilfe von Tools zur statischen Code-Analyse entdecken. Diese sind aber schwer zu beseitigen, besonders wenn die Zeit eher in das bessere Aussehen des Programms investiert wird. Komplexe Änderungen führen zur Entstehung von Kettenreaktionen, denen Programmieranfänger nicht gewachsen sind. Darunter leidet an erster Stelle die Motivation der Studierenden. Dies soll vermieden werden. In Rahmen eines modellbasierten Entwicklungsprozesses lässt sich hohe Code-Qualität von Anfang des Projektes an integrieren. Aus diesem Grund werden neben der Thematisierung der Code-Qualität in den Einführungsveranstaltungen, der Messung mit PMD und dem Beseitigungszwang die Modelle näher analysiert, um Mängel frühzeitig zu erkennen und zu beseitigen. Wir haben Merkmale im Modell identifiziert, die direkt zu Mängeln im Programm-Code führen, wie schlecht gewählte Bezeichner und lange Parameterlisten. Andere Merkmale liefern uns Hinweise darauf, welche Methoden und Klassen zu lang und zu komplex

werden könnten.

Tom DeMerko sagt, dass die Alternative zur Fehlerbeseitigung die Fehlerlosigkeit ist [De04]. Deswegen beschäftigen wir uns mit der Integration von Qualität Im Rahmen eines Software-Entwicklungsprozesses von Anfang an.

Literaturverzeichnis

[As15] Astah, http://astah.net/, abgerufen am 21.10.2015.

[Ba94] Basili, V. R., Caldiera, G., Rombach, H. D.: The Goal Question Metric Paradigm, In: Encyclopedia of Software Engineering - 2 Volume Set, John Wiley & Sons, 1994.

[CK94] Chidamber, S. R., Kemerer, C. F.: A Metrics Suits for Object Oriented Design. IEEE Transaction on Software Engineering, 20(6), 476-493, June 1994

[De04] DeMarco, T: “Was man nicht messen kann, kann man nicht kontollieren”, mitp-Bonn, 2004.

[Fo99] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA., 1999

[Ga09] Gamma, E., Helm, R., Johnson, R.: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley; 2009

[LM06] Lanza, M., Marinescu, R., Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the design of Object-Oriented Systems, Springer, 2006.

[Ma09] Martin, R. C.: Clean Code, Prentice Hall, 2009.

[Mc76] McCabe, Thomas: A complexity measure. In: IEEE Transaction on Software Engineering, Nr. 4, S. 308-320, 1976.

[PMD15] PMD, https://pmd.github.io/, abgerufen am 21.10.2015.

[Re14] Remmers, J., Code-Qualität im Software-Praktikum, Bachelorarbeit, Fakultät für Informatik, TU Dortmund, 2014.

[Ro70] Royce, W. W.: Managing the Development of Large Software Systems: Concepts and Techniques - WESCON Conference Proceeding, August 1970

[Ru12] Rupp, C., Queins, S. und die SOPHISTen, Nürnberg. UML 2 glasklar: Praxiswissen für die UML-Modellierung, Carl Hanser Verlag, München, 2012.

[Sc15] Schmedding, D., Vasileva, A., Remmers, J.: Clean Code - ein neues Ziel im Software-Praktikum, SEUH 2015, Dresden, 2015

[SP15] Software-Praktikum, https://sopra.cs.tu-dortmund.de/wiki/, abgerufen am 21.10.2015.

[TC02] Tang, M. H., Chen, M. H.: Measuring OO Design Metrics from UML, UML 2002, Springer-Verlag Berlin Heidelberg, 2002

[Va15] Vasileva, A., Schmedding, D.: Integration von Qualitätsaspekten in einen Entwicklungsprozess, In Proceeding of Metrikon 2015, Köln, 2015

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 61

On the de-facto Standard of Event-driven Process Chains: How EPC is defined in Literature

Dennis M. Riehle[1], Sven Jannaber[2], Arne Karhof[2], Oliver Thomas[2], Patrick Delfmann[1], Jörg Becker[3]

Abstract: The Business Process Modelling Notation (BPMN) and the Event-driven Process Chain (EPC) are both frequently used modelling languages to create business process models. While there is a well-defined standard for BPMN, such a standard is missing for EPC. As a standard would be beneficial to improve interoperability among different vendors, this paper aims at providing the means for future EPC standardization. Therefore, we have conducted a structured literature review of the most common EPC variants in IS research. We provide a structured overview of the evolution of different EPC variants, describe means and capabilities and elaborate different criteria for decision-making in regard to including EPC variants in a standardization process.

Keywords: event-driven process chain, EPC, process modelling, literature review, EPC variants, EPC dialects, exchange formats, EPC evolution, standardisation

1 Process Modelling with Event-driven Process Chains

To support the management of business processes, a multitude of business process modelling languages (BPML) has emerged over time, for example BPMN or the Unified Modelling Language (UML) [Aa13]. As a result of the growing interest of researchers and practitioners, many BPML have been standardized by respective standard development organizations (SDO) [KLL09]. One of the most dominant languages for business process modelling is the EPC developed in 1992. Consequently, the EPC has been extensively researched and is still ongoing subject of discussion in the business process management (BPM) domain [Fe09, Fe13, HFL09, Ri00]. However, despite its maturity, no attempts for a successful EPC standard-making have been undertaken to this day. As a consequence, EPC loses ground compared to other languages regarding diffusion, usage and acceptance [DKK14, Fe13, KLL09].

A BPML standard typically contains constructs such as modelling elements, a (formal) syntax, a meta-model or a model exchange format [e.g. Om11]. Ultimately, a standard ensures international adherence to those constructs and not only serves as an agreed-upon

[1] University of Koblenz-Landau, Institute for Information Systems Research, Universitätsstraße 1, 56016, Koblenz, riehle@uni-koblenz.de, delfmann@uni-koblenz.de
[2] University of Osnabrück, Institute for Information Management and Information Systems, Katharinenstraße 3, 49069, Osnabrück, arne.karhof@uni-osnabrueck.de, sven.jannaber@uni-osnabrueck.de, oliver.thomas@uni-osnabrueck.de
[3] University of Münster, European Research Center for Information Systems, Leonardo Campus 3, 48149, Münster, becker@ercis.uni-muenster.de

basis for further language refinement, but also facilitates applicability in practice, e.g. by increasing the interoperability of process models [Fo03, MN06]. Although there exist numerous publications that propose specifications for constructs of the EPC language [e.g. MA07, NR02, Ro08], a unified approach towards EPC standardization has not been initiated to date [KLL09]. The challenges of systematic standardization primarily originate in the large amount of scientific contributions that have been published, which have significantly increased the variety of EPC-related propositions. Hence, identifying and determining what language constructs and extensions to include in an EPC standard becomes a difficult task.

The paper at hand aims at providing the groundwork for a successful EPC standardization. The characterized obstacles are tackled by reviewing, ultimately synthesizing and evaluating relevant scientific work in the field of EPCs. Since standard-making heavily depends on agreement and consensus of a domain community [DG90, FKL03], an overview over state-of-the-art in EPC research is necessary to create a common understanding of the EPC language. The main focus of this paper is hereby put on the evolution of the EPC language, more specifically on the various language variants that have been developed over time, which extend the basic EPC language with additional concepts and constructs. We believe that providing transparency over a language's natural evolution is needed for the establishment of a common ground that supports a subsequent successful standard-making procedure.

The conducted literature review has been able to examine the evolution of the EPC in terms of variants and included concepts that have emerged over the last years. In doing so, the paper extends the body of knowledge by providing an overview of the EPC language progression over time. Furthermore, relevant language extensions are introduced and evaluated according to predefined criteria, ultimately resulting in a consolidation of previous work that is able to serve as a basis for an EPC standard. Finally, the paper proposes a suggestion of constructs and extensions to consider for an EPC standard.

The paper is structured as follows. Section 2 introduces basic concepts of the EPC language. Additionally, the evolution of related business process management languages is highlighted briefly. In Section 3, the applied research methodology is characterized. The findings of the literature review are presented in Section 4 and discussed in Section 5. The paper concludes with a summary of the results with respect to the research question and an outlook indicating further work.

2 Theoretical Background and Related Work

At the time when EPCs emerged in the 1990s from a joint work of the Institute for Information Systems in Saarbrücken and SAP [KNS92], the effort in BPM standardization has been a negligible factor. Actually, the first standard published in this particular field has been the Workflow Reference Model by the Workflow Management Coalition and was first released three years after the EPC, in 1995 [Ho95]. If we instance the year 2004,

in which the BPMN has been publically released, there have been more than ten organizations, who developed standards in the field of BPM [NM06]. Furthermore, not only the quantity of standardization endeavours has been smaller, but also the extent significantly differed from nowadays standards. While the initial standard of the Workflow Reference Model has been described on approximately 50 pages, the BPMN 2.0 standardization paper goes beyond the constraint of 500 pages [Om11]. Despite this heavy increase of extent, the generally recognized need for standardization in 2004 led to the development of a BPMN standard in only two years, while after more than one decade there is still no received standard for EPC models.

Besides the lack of a de-jure standard for EPC models, there is another ancillary effect ensuing from the historical background – as there never has been a coalition, a committee or any other form of society establishing a standardization process and additionally discussing possible extensions of the EPC, an equivalent amount of proposals to alter the EPC have been developed and published. For partial overviews of these extensions, the reader can refer to [Me08] and [SDL05]. Unfortunately, these papers do not provide insights to the evolution of the EPC-extensions, their dependencies among themselves, or detailed descriptions. Also, the papers' main issues do not concern EPC extensions themselves. Therefore, they are just mentioning or briefly describing them. In consequence of this heterogenous area of EPC interpretations, it is necessary to gain a structured overview of these versions and to cover the mentioned aspects above. Only then it is possible to completely capture all relevant aspects for an EPC standard. There are some publications, which already address partial areas of EPC standardization, either by proposing formal notations of the EPC [Aa99, Me08] or by assembling informal rules for EPC modelling [e.g. Fe13]. Other publications have analysed file-based exchange formats for EPC models [Ri16] and the implementation of EPC in different modelling tools [Ka16]. Unfortunately, such papers neglect the occurrence of EPC-variations and do not discuss their relevance.

In favor of a better understanding of the proposed alterations which are given in section 4, we briefly introduce the initial EPC elements as introduced by [KNS92]. This first version of the EPC only consisted of events, functions and operators. Thereby, the event is defined as an "occurrence of a defined condition", while the function is a "process that converts an input state to an output state". Beginning with events, these entities always have to alternate. Furthermore, the initial EPC provided operators for splitting and joining the control flow of a process model. These operators were distinguished as conjunctive, disjunctive and adjunctive and corresponded to AND, XOR and OR operators respectively.

3 Methodology

We have conducted a structured literature review in the discipline of information systems, as suggested by [WW02] and [Br13]. Since the first publication regarding the EPC was published as a working paper in the working paper series of the institute for information

systems at the University of Saarbrücken, we included their working series with a total of 198 papers as a data source in our review process. Additionally, we considered the EPC workshop from 2002 to 2009, whose proceedings with a total of 57 papers were included as our second data source. Lastly, we queried two scientific search engines with the terms "Ereignisgesteuerte Prozesskette" and "event-driven process chain" (in several spellings) to find further publications. We are aware that these generic search queries deliver a huge result set, however we think that there is no simple and more specific query that still includes all publications of different EPC adoptions in different application areas. Altogether, our queries returned 1.806 publications at SpringerLink[4] and 198 publications at ScienceDirect[5]. The large difference is due to SpringerLink including more German publications than ScienceDirect.

Due to the large number of 2.259 publications, we followed [WW02] in considering only titles first and discarded papers, whose titles suggested an application of EPC without contributing to the EPC modelling language itself. Further, for technical reasons, we excluded papers of which we could neither find a full text online nor in the libraries of three different universities. This left us with 316 papers. After removing duplicates, reading the abstracts and where necessary the contents as well, we came to a set of 79 publications in our first iteration.

Next, we conducted a forward and backward search to find additional literature that might not have been found by our search engines. While the backward search was done manually, we used Google Scholar[6] for the forward search. In this iteration we also revised some of our decisions on discarding papers in the previous iteration, e.g. in case a referenced paragraph generated more insight to the papers' content than solely its title. After reading the abstracts of the papers delivered by forward and backward search, and after another removal of duplicates, we found 35 new publications. Therefore, we came to a final set of 114 papers.

4 Different EPC Variants from Literature

More than 20 years ago, the idea of modelling business processes with alternating events and functions was published by [KNS92]. Not surprisingly, event-driven process chains have heavily developed over the years, leading to similar, yet different understandings of EPC in literature. Many authors have suggested extending the EPC in one or the other way and several formalizations of the EPC modelling language have been provided.

Within our set of 144 publications regarding EPCs, we found 14 different variations of the EPC, which we will call EPC dialects. For all these 14 different dialects, the authors described in one or more papers how their variation of the EPC is defined and in which

[4] http://link.springer.com/
[5] http://sciencedirect.com/
[6] http://scholar.google.com/

terms they modified the base EPC by [KNS92]. An overview of these 14 EPC dialects is provided in Tab. 1. Additionally, we identified 8 different exchange formats for EPC models (counting different versions of the same format separately, due to incompatibilities), which are capable of storing EPC models of one or more of the 14 different EPC dialects.

Name	References	Short Abstract
EPC	[KNS92]	The prototype of all event-driven process chains, consisting only of functions, events and connectors
Extended EPC (**eEPC**)	[HKS93] [GS94] [KT97]	An enriched EPC, including organizational units, information objects, IT systems and process refinements
Real-Time EPC (**rEPC**)	[HWS93]	Includes a state machine, which is modelled in parallel to the EPC, events and functions of the EPC are linked to conditions and actions of the state machine
EPC*	[ZR96]	Information objects can be connected to model data flow and conditions can be added to the control flow, e.g. to start events
Object-oriented EPC (**oEPC**)	[SNZ97] [NZ98]	Uses an object-oriented process definition, where activities are carried out on business objects, therefore such objects are modelled instead of traditional functions
Risk EPC	[BO02]	Adds a risk element to the EPC which can be linked with function to visualize potential business risks
Fuzzy EPC	[THA02] [TD06] [Th09]	A new fuzzy connector allows to model fuzzy decision-making in business processes, using variables and decision rules related to the fuzzy connector
Yet another EPC (**yEPC**)	[MNN05]	Extends the EPC to support state-based workflow patterns, multi-instantiation and cancellation
Risk EPC extended	[RM05] [RW08]	Adds different risk events which are triggered when an exception occurs, further risk-management functions can be used to handle the exception
Configurable EPC (**C-EPC**)	[RA07] [Re05]	Functions, events and connectors are configurable, several EPCs can be generated from a single EPC reference model
Semantic EPC	[TF06] [FKS09]	EPC elements are linked to ontologies, to improve semantics and prevent ambiguity
Nautilus EPC (**N-EPC**)	[KUL06]	Allows modelling of several events between two functions, trivial events can make decisions and can hence be followed by an XOR or OR connector
Service EPC	[HW07]	Traditional functions are replaced by services, which can be synchronous or asynchronous, introduce timeouts and message events
Configurable integrated EPC (**C-iEPC**)	[Ro08]	Extends the configuration introduced in the C-EPC to information objects, IT systems and organizational objects, allowing an even greater range of configuration

Tab. 1: Overview of EPC dialects most prominent in literature

While the original EPC only considered the process flow by using events and functions,

this basic EPC model was soon extended. A very early extension was the real-time extension to EPC, which included a state machine similar to Petri nets. Events and function of the EPC are linked to conditions and actions of the state machine, enabling a state-based execution of rEPC models [HWS93].

Another early extension of the EPC is referred to as extended EPC (eEPC) in literature. Extended EPCs include organizational units, which are connected to functions to model process responsibilities, information objects, which represent abstract data (documents) and can be used or generated by functions, IT systems, which can be used by functions, and process refinements, which aggregate a set of activities, i.e. a subordinate EPC to a single element [GS94, HKS93, KT97, Ro96]. The eEPC was widely accepted and all further dialects are based heron.

The EPC* extension is provided by [ZR96], who tried to overcome the shorting of traditional EPCs only representing the control flow in a process. Therefore, EPC* introduced relations between information objects to represent the data flow within a single process. Furthermore, conditions can be added to control flows to enable decision-making, and organizational units can be connected with start events to model organizational units that are allowed to initiate the process.

If processes are not understood as a set of activities (function-oriented process definition), but rather as operations on a business object (object-oriented process definition), an object-oriented EPC (oEPC) can be used to model business processes [NZ98, SNZ97]. The oEPC replaces functions which object classes, which includes methods for operating on the business object. Still, organizational units and information objects can be attached to objects.

An idea of explicitly modelling business risks in EPC models was first suggested by [BO02], who add a risk element, which when connected with a function indicates that this function can cause a potential problem. We refer to this dialect as the “Risk EPC”. The idea of modelling risks has been further adapted by [RM05] and [RW08], who propose to separate between different kinds of risks. [RW08] suggest three different risk elements, which can be triggered by functions when an exception occurs. These risk events can be followed by different kinds of risk handling functions. This dialect, which we refer to as “Risk EPC extended”, allows not only to model different kinds of risks, but also to model how unexpected situations are handled in the process.

With the Fuzzy EPC, one is capable of modelling fuzzy decision making based on textual variables and decision rules, for which [THA02] introduce the fuzzy connector. Further publications describe attributes and details of Fuzzy EPCs in more detail [TD06, Th09]. Similarly, the Yet another EPC (yEPC) introduces a new connector as well, namely the empty connector. The yEPC further adds process parameters for multi-instantiation and a cancellation area for cancellation support [MNN05], to make the EPC capable of executing common workflow patterns described in [Aa03].

The suggestion of using configurable process models as a basis for reference modelling

[Aa06] lead to the development of a configurable EPC (C-EPC). The C-EPC includes configurable functions, events and connectors which make the control flow of the EPC configurable [RA07, Re05]. With this technique, several different concrete EPC models can be generated from a single C-EPC, depending on the configuration. This approach has been further extended by [Ro08], who specified configuration for organizational units, information objects and IT systems. This so-called configurable integrated EPC (C-iEPC) enables extremely configurable scenarios and therefore a high reusability of EPC models.

A simplified dialect called nautilus EPC (N-EPC) has been proposed by [KUL06]. Contrasting to other EPC dialects, trivial events are able to make a decision. Therefore, trivial events can be followed by an XOR or an OR operator. Technically, this dialect omits the function succeeding trivial events, as this function is also regarded to be trivial.

Another fairly recent publication adopts the idea of IT services and regards business processes as services, which are consumed during the process execution. Consequently, [HW07] add functions, which refer to service calls that can either be synchronous or asynchronous. We refer to this dialect as "service EPC". Since the introduction of asynchronous functions requires some kind of message processing when an asynchronously executed function finishes, the service EPC also introduced events to be triggered upon message receipt. This allows a service EPC to start the execution of several asynchronous services first, and then to wait until the service execution finishes.

Besides these 14 different EPC dialects described in literature, we also found different exchange formats. Exchange formats provide a computer-readable data storage for EPC models, which helps to store, transfer and reuse EPC models in different environments. The specification of an exchange format provides an implicit definition of an EPC dialect, by the EPC elements which have been implemented in the concrete exchange format. For all 8 different exchange formats we have identified in literature, we have analysed with which of the 14 different EPC dialects they are compatible with. The evolution of EPC dialects and exchange formats over time is shown in Fig. 1.

A first XML notation for EPC models (XML EPC) has been proposed by [GK02], who provide an XML schema to describe an eEPC, including functions, events, connectors, information objects, organizational units and process refinements. At the same time, a similar yet incompatible XML schema has been described by [MN02], who introduce the EPC Markup Language (EPML) as a general purpose format for exchanging EPC models [MN04].

[WS06] use a different approach by adopting the Graph Exchange Language (GXL) for use with EPC models. GXL is capable of storing any conceptual models as graphs, using nodes to reflect conceptual elements and edges to reflect relations. Though [WS06] do not provide an example for all elements of the eEPC, their exchange format is capable of representing all eEPC elements, as new node types can easily be defined using a custom identifier.

For semantic EPCs, [FKS09] have proposed an ontology based exchange format. By

transforming elements of the semantic EPC to ontology concepts, they embed the EPC model within an ontology. We refer to this approach as “XML EPC Ontology”. Similar to the GXL approach, this leads to storage formats where EPC elements are only meaningful by their description, which usually is a string-based identifier. However, for automated processing it is helpful, if EPC elements are defined within the specification on an exchange format.

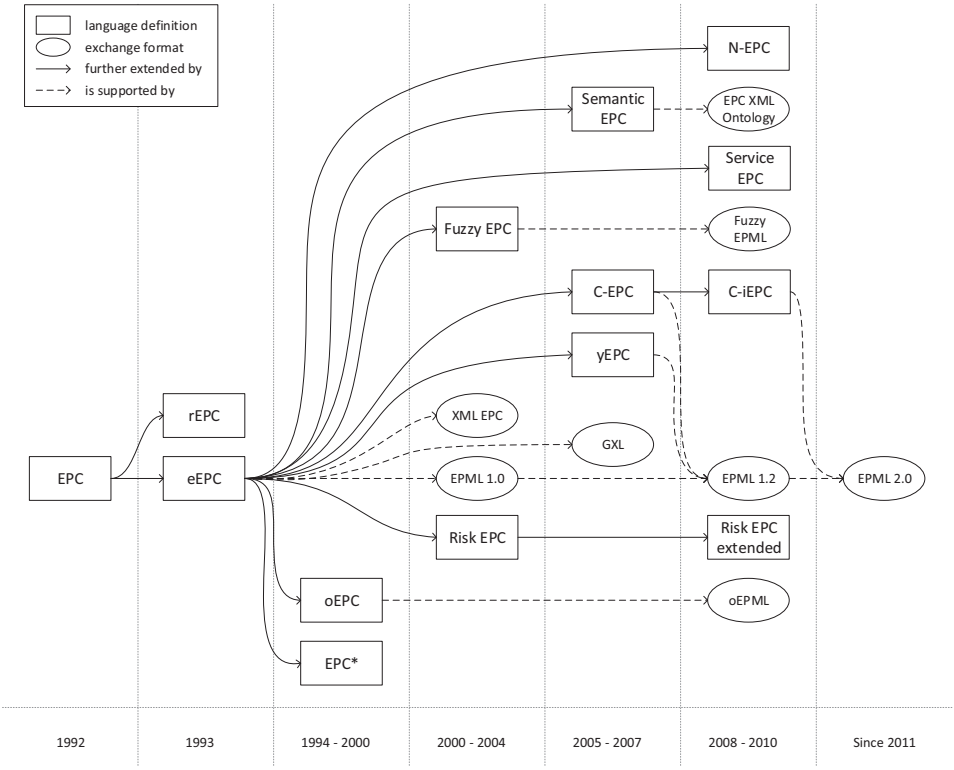


Fig. 1: Evolution of different EPC dialects and exchange formats over time

Besides these four different and independently developed formats, the EPML approach by [MN02] has been adapted several times. A modification called oEPML has been developed by [Ho09], who extends the XML schema for objects of the oEPC dialect. Similarly, [TD08] suggested a modification named Fuzzy EPML, which adds a fuzzy connector to the EPML schema and therefore is capable of representing Fuzzy EPC models. However, Fuzzy EPML and oEPML have not been integrated yet, making these two modifications of EPML incompatible to each other. As a consequence, there is no exchange format that can be used for eEPC, Fuzzy EPC and oEPC models altogether.

After a modification for EPML to support C-EPC models was suggested by [Me05], an updated schema for EPML has been released as EPML 1.2 [Me09], which supports yEPC

and C-EPC model. Additionally, with EPML 2.0 [Me11], support for C-iEPC models was added, which makes EPML 2.0 capable of storing eEPC, yEPC, C-EPC and C-iEPC models. From the number of supported EPC dialects, EPML 2.0 supports the largest variety of EPC dialects.

Summing up the results of our structured literature review, we found 14 different EPC dialects and 8 different exchange formats. Most EPC dialects developed in the last 20 years are based on the eEPC dialect, therefore these dialects share a common understanding of not only events, functions and connectors, but also of information objects, IT systems and organizational units.

There are some extensions, which we have not considered as EPC dialects, for example the SEQ connector [Pr95] or the ET and OR1 connector [Ro96]. These connectors focus on providing a simplification for modelling and can be added to any EPC dialect. Additionally, the SEQ, ET and OR1 connectors can be converted into structures consisting of eEPC elements only, i.e. functions, events and XOR, OR and AND connectors [Ru99, p.62 ff.]. Furthermore, we have neither considered modified EPCs (modEPC) nor agent-oriented EPCs (xEPC), as these EPC dialects were only briefly discussed at one single conference and in working papers of the years 1999 and 2000. Unfortunately, we were not able to obtain a full-text as described in the methodology section before.

5 Towards an EPC Standard

The diversity of EPC dialects and exchange formats calls for a standardization procedure to develop a specification for an integrated EPC language and exchange format. As mentioned in our introduction, a standardized modelling language can be more easily implemented by tool vendors and hence can spread faster in the BPM community.

It is worth mentioning that an EPC standard would not only be beneficial for companies that start modelling business processes with event-driven process chains, but also for companies that already have modelled EPCs in the past and still store these models as legacy models. With an EPC standard, such legacy models could be reused in any modelling environment supporting the prospective EPC standard. Possibly some legacy models have to be revised, but it is safe to assume that basic EPC construct will find their way into the standard. Moreover, procedures for migrating one modelling language to another, e.g. EPC to BPMN as suggested by [DT09], could be implemented independently of a modelling tool.

While the benefits of an EPC standard are numerous and rather obvious, there are several open questions yet to be answered. Most importantly, it has to be decided which EPC dialects should be part of an EPC standard. There are several criteria that one might consider. On the one hand, one could evaluate the dominance of an EPC dialect in the literature, e.g. for the rEPC there is – to the best of our knowledge – only one publication [HWS93], which might be an indicator for the rEPC being less relevant to researchers. On

the other hand, one might consider the compatibility of different EPC dialects. For example, the N-EPC breaks with the formal semantics of traditional EPCs, as it allows trivial events to make decisions, which is not allowed in any other EPC dialect presented in this paper. Therefore, one might exclude the N-EPC for compatibility reasons.

Besides regarding EPC dialects in the literature, one might also consider the spread of EPC dialects in practice. Since an EPC standard should be implemented and used by practitioners, such a standard should include all artefacts which are needed in practice. Therefore, one should perform a market analysis of BPM modelling tools to get an understanding of EPC dialects which are commonly used in practice. The results of such a market analysis should be included in the process of deciding which EPC dialects are becoming parts of an EPC standard.

Since practitioners can only use the EPC dialects which have been implemented in EPC modelling tools, there might be a discrepancy between what practitioners would like to see in an EPC model and what they can actually model with their modelling tool. Therefore, a survey among practitioners might deliver further insights, especially in terms of useful EPC dialects which have not made it into modelling tools yet.

To further circumstantiate the integration or exclusion of certain EPC dialects and artefacts in an EPC standard, one could interview EPC experts. Interviews with individuals might deliver more detailed results than a survey and provide better arguments for decision-making. Together with a group of experts, a final decision on what to be included in an EPC standard should be made. This should be done once modelling tools have been analysed and practitioners have been surveyed.

For a first step towards standardization, we have analysed the type of EPC extension, the type of specification of that extension, the impact in research and the degree of generalization for each 14 EPC dialects. An overview is provided in Tab. 2. To estimate the impact in research, we have used Google Scholar to conduct a forward search for each paper and to count the number of citations each paper has received (column “Cites” in Tab. 2). For EPC dialects that consist of multiple papers, we have conducted a forward search for each paper, and have built an outer set, i.e. eliminated all duplicates in the citations. That number is listed in brackets in the column “Impact”. We have classified the impact as “high” if there were more than 100 citations, “noticeable” if there were more than 50 citations, “medium” if there were more than 25 citations and “low” otherwise.

In regard to the way an EPC dialect was specified, we distinguished between an enumerative specification, i.e. the authors provided a textual or visual list of all elements or extensions they provided, a formal semantic specification, i.e. a specification in terms of mathematical equations based on set theory and a meta-model based specification, where the authors provided a meta-model for their EPC variant.

The column “Type of Extension” in Tab. 2 lists the means of each EPC variant, for example if the authors provide additional elements to enrich the information of an EPC model or if the authors aim at a different goal like achieving execution of workflows.

Lastly, the column "Generalizable" describes whether the variant is applicable to business process models modelled in EPC in general or if the dialect aims at a certain application domain. As the N-EPC breaks with formal semantics of other EPC dialects, it is not generally applicable to other EPC models.

Dialect	Reference	Cites	Impact	Specification	Type of Extension	Generalizable
EPC	[KNS92]	993	high	enumerative	-	generally applicable
eEPC	[HKS93]	67	high (163)	enumerative	additional elements	generally applicable
	[GS94]	19		enumerative		
	[KT97]	84		enumerative		
rEPC	[HWS93]	10	low	formal semantics	execution of processes	generally applicable
EPC*	[ZR96]	35	medium		execution of workflows	generally applicable
oEPC	[SNZ97]	64	noticeable (74)	enumerative	additional elements	for modelling business objects
	[NZ98]	18		enumerative		
Risk EPC	[BO02]	25	low	enumerative	additional element	generally applicable
Risk EPC extended	[RM05]	150	high (166)	enumerative	additional elements	generally applicable
	[RW08]	17		meta-model		
Fuzzy EPC	[THA02]	24	medium (34)	meta-model	additional elements and decision tables	generally applicable
	[TD06]	10		formal semantics		
	[Th09]	9		meta&formal		
yEPC	[MNN05]	41	medium	enumerative	execution of workflows, additional elements	generally applicable
C-EPC	[RA07]	505	high (526)	formal semantics	reference modelling	generally applicable
	[Re05]	42		enumerative		
Semantic EPC	[TF06]	5	low (22)	enumerative	annotation w/ontologies	generally applicable
	[FKS09]	17		enumerative		
N-EPC	[KUL06]	22	low	formal semantics	simplification	conflicts with formalization of common EPC dialects
Service EPC	[HW07]	8	low	enumerative	additional elements	for modelling business services
C-iEPC	[Ro08]	67	noticeable	formal semantics	extended configuration	generally applicable

Tab. 2: Attributes of different EPC dialects for decision-making in a standardization process

Besides fostering a common understanding of EPC elements, an EPC standard should also provide an exchange format for EPC models to improve interoperability among different modelling tools. We think that such an exchange format should be focused on EPC models only and not on process models in general. Approaches like GXL, which can be applied to different process modelling languages as EPC or BPMN, are more complex, thus harder to implement for tool developers and, in most cases, do only support portions of modelling languages, namely those concepts which are supported by all considered languages. Contrastingly, an EPC-specific exchange format like EPML has a well-defined set of element types it can represent, namely those of the underlying EPC dialect. This makes adoption of such an exchange format in modelling tools much easier.

An exchange format for EPC models should be designed carefully. If an EPC standard is extensible, as discussed above, an exchange format needs to be extensible as well. Therefore, an exchange format should be designed in a way that additional information can be added without breaking backward compatibility, so that a tool, which can import a basic EPC model, can also import an EPC model with extensions it is not aware of. Of course, such extensions the modelling tool is not aware of will not be displayed or in any way handled by the modelling tool. The important point is that a modelling tool needs to recognise information in the exchange format it does not know as an unknown EPC extension that has not been implemented in the tool. Then, the modelling tool can simply ignore the additional information. Obviously, in this scenario, an EPC model might become incomplete if imported in a modelling tool which does not implement all EPC extension used within the EPC model. However, this way, a modelling tool could still import as much as possible from a model, providing the greatest possible benefits to users in terms of reusing models.

6 Conclusion

In this paper, we have performed a structured literature review on business process modelling with event-driven process chains and have identified 14 different EPC dialects in the literature. While an overview of EPC dialects has already been provided in [Me08, p.28 ff.] and [SDL05], we have found more EPC dialects and we have described all EPC dialects we found in more detail. Additionally, we have described the evolution of different EPC dialects over time, which – to the best of our knowledge – has not been done to this day. Therefore, we provide a more recent and greater overview of EPC variants.

Furthermore, we have considered exchange formats for EPC models, of which we have found 8 different exchange formats in literature. We have related these 8 different exchange formats to the 14 different EPC dialects we found and have analysed, which exchange format can be used for storing EPC models of which dialect. With this analysis, we have shown that for some EPC dialects there are no well-known exchange formats and that there are several incompatible exchange formats. Only a few exchange formats support more than one EPC dialect.

Lastly, we have analysed the 14 different EPC dialects in four different criteria, their impact in research, the type of specification, the type and mean of the variant and the generalizability. By this, we have shown that some EPC variants can be considered being more prominent than others and therefore being more adequate for inclusion in a future EPC standard.

With our research, we have contributed towards the development of a standardized EPC modelling language and a standardized EPC exchange format, shortly referred to as an EPC standard. While a standard is beneficial for research and practice under several aspects, such a standard has not been developed yet. With our research, we have provided parts of a research agenda towards a successful EPC standardization.

Acknowledgement: The research presented in this paper is part of the SPEAK project and is funded by the Federal Ministry for Economic Affairs and Energy (BMWi) under grant number 01FS14030.

References

[Aa03] Aalst, W.M.P. van der, Hofstede, A.H.M. ter, Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases 14, pp. 5–51. 2003.

[Aa06] Aalst, W.M.P. van der, Dreiling, A., Gottschalk, F., Rosemann, M., Jansen-Vullers, M.H.: Configurable Process Models as a Basis for Reference Modeling. In: C.J. Bussler, A. Haller (eds.) Business Process Management Workshops - BPM 2005 International Workshops. pp. 512–518. Springer, Berlin 2006.

[Aa13] Aalst, W.M.P. van der.: Business Process Management: A Comprehensive Survey. ISRN Software Engineering 2013, pp. 1–37. 2013.

[Aa99] Aalst, W.M.P. van der.: Formalization and verification of event-driven process chains. Information and Software Technology 41, pp. 639–650. 1999.

[BO02] Brabänder, E., Ochs, H.: Analyse und Gestaltung prozessorientierter Risikomanagementsysteme mit Ereignisgesteuerten Prozessketten. In: EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens, pp. 1–5. 2002.

[Br13] Brocke, J.M. vom, Simons, A., Niehaves, B., Riemer, K., Plattfaut, R., Cleven, A.: Reconstructing the giant: On the importance of rigour in documenting the literature search process. In: 17th European Conference on Information Systems, pp. 1–13. Verona, Italy 2013.

[DG90] David, P.A., Greenstein, S.: The Economics of Compatibility of Standards: An Introduction to Recent Research. Economics of Innovation and New Technology 1, pp. 3–41. 1990.

[DKK14] Drawehn, J., Kochanowski, M., Kötter, F.: Business Process Management Tools 2014. Stuttgart, Germany 2014.

[DT09] Decker, G., Tscheschner, W.: Migration von EPK zu BPMN. In: EPK 2009. 8. Workshop der Gesellschaft für Informatik e.V. (GI) und Treffen ihres Arbeitkreises "Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK). Gesellschaft für Informatik, pp. 91–109. 2009.

[Fe09] Fettke, P.: Ansätze der Informationsmodellierung und ihre betriebswirtschaftliche

Bedeutung: Eine Untersuchung der Modellierungspraxis in Deutschland. Schmalenbachs Zeitschrift für betriebswirtschaftliche Forschung (zfbf) 61, pp. 550–580. 2009.

[Fe13] Fellmann, M., Bittmann, S., Karhof, A., Stolze, C., Thomas, O.: Do We Need a Standard for EPC Modelling? The State of Syntactic, Semantic and Pragmatic Quality. In: 5th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA), pp. 103–116. St. Gallen, Switzerland 2013.

[FKL03] Fomin, V., Keil, T., Lyytinen, K.: Theorizing about standardization: integrating fragments of process theory in light of telecommunication standardization wars. Sprouts: Working Papers on Information Environments, Systems and Organizations 3, pp. 29–60. 2003.

[FKS09] Filipowska, A., Kaczmarek, M., Stein, S.: Semantically annotated EPC within semantic business process management. In: Lecture Notes in Business Information Processing, pp. 486–497. Milano, Italy 2009.

[Fo03] Fomin, V.: The Role of Standards in the Information Infrastructure Development. MISQ Special Issue Workshop 1993, pp. 302–313. 2003.

[GK02] Geissler, M., Krüger, A.: Eine XML-Notation für Ereignisgesteuerte Prozessketten (EPK). In: Workshop der Gesellschaft für Informatik e.V. (GI) und Treffen ihres Arbeitskreises "Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK)," pp. 81–86. Trier, Germany 2002.

[GS94] Galler, J., Scheer, A.-W.: Workflow-Management - Die ARIS-Architektur als Basis eines multimedialen Workflow-Systems. In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 108, Universität des Saarlandes 1994.

[HFL09] Houy, C., Fettke, P., Loos, P.: Stilisierte Fakten der Ereignisgesteuerten Prozesskette – Anwendung einer Methode zur Theoriebildung in der Wirtschaftsinformatik. In: EPK 2009. 8. Workshop der Gesellschaft für Informatik e.V. (GI) und Treffen ihres Arbeitkreises "Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK). Gesellschaft für Informatik, pp. 22–41. Bonn, Germany 2009.

[HKS93] Hoffmann, W., Kirsch, J., Scheer, A.-W.: Modellierung mit Ereignisgesteuerten Prozeßketten. In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 101, Universität des Saarlandes 1993.

[Ho09] Hogrebe, F., Nüttgens, M., Kern, H., Kühne, S.: Towards an Integrated Product and Process Modelling : oEPC Markup Language (oEPML) for object-oriented Event-driven Process Chains (oEPC). In: Informatik 2009: Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2009.

[Ho95] Hollingsworth, D.: The Workflow Reference Model. Management am, pp. 1–55. 1995.

[HW07] Huth, S., Wieland, T.: Geschäftsprozessmodellierung mittels Software-Services auf Basis der EPK. In: V. Nissen, M. Petsch, H. Schorcht (eds.) Service-orientierte Architekturen. pp. 61–76. Deutscher Universitäts-Verlag, Wiesbaden 2007.

[HWS93] Hoffmann, W., Wein, R., Scheer, A.-W.: Konzeption eines Steuerungsmodells für Informationssysteme–Basis für die Real-Time-Erweiterung der EPK (rEPK). In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 106, Universität des Saarlandes 1993.

[Ka16] Karhof, A., Jannaber, S., Riehle, D.M., Thomas, O., Delfmann, P., Becker, J.: On the de-facto Standard of Event-driven Process Chains: Reviewing EPC Implementations in Process Modelling Tools. In: Proceedings of the Modellierung 2016, Karlsruhe, Germany 2016.

[KLL09] Ko, R.K.L., Lee, S.S.G., Lee, E.W.: Business process management (BPM) standards: a survey. Business Process Management Journal 15, pp. 744–781. 2009.

[KNS92] Keller, G., Nüttgens, M., Scheer, A.-W.: Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)." In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 89, Universität des Saarlandes 1992.

[KT97] Keller, G., Teufel, T.: SAP R/3 prozeßorientiert anwenden. Addison-Wesley, Bonn 1997.

[KUL06] Kopp, O., Unger, T., Leymann, F.: Nautilus Event-driven Process Chains: Syntax, Semantics, and their mapping to BPEL. In: Proceedings of the 5th GI Workshop on Event-Driven Process Chains (EPK 2006), pp. 85–104. 2006.

[MA07] Mendling, J., Aalst, W.M.P. van der.: Formalization and Verification of EPCs with OR-Joins Based on State and Context. Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE'07) 4495, pp. 439–453. 2007.

[Me05] Mendling, J., Recker, J., Rosemann, M., Aalst, W.M.P. van der.: Towards the Interchange of Configurable EPCs: An XML-based Approach for Reference Model Configuration. In: Proceedings Workshop Enterprise Modelling and Information Systems Architectures, pp. 8–21. 2005.

[Me08] Mendling, J.: Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness. 2008.

[Me09] Mendling, J.: EPML Schema 1.2. 2009. Available at: http://www.mendling.com/EPML/EPML_12.xsd, accessed 15. Oct 2015.

[Me11] Mendling, J.: EPML Schema 2.0. 2011. Available at: http://www.mendling.com/EPML/EPML_2.0.xsd, accessed 15. Oct 2015.

[MN02] Mendling, J., Nüttgens, M.: Event-Driven-Process-Chain-Markup-Language (EPML): Anforderungen zur Definition eines XML-Schemas für Ereignisgesteuerte Prozessketten. In: Workshop der Gesellschaft für Informatik e.V. (GI) und Treffen ihres Arbeitskreises "Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK)"2, pp. 87–94. 2002.

[MN04] Mendling, J., Nüttgens, M.: XML-based Reference Modelling: Foundations for an EPC Markup Language. In: J. Becker, P. Delfmann (eds.) Referenzmodellierung. pp. 51–72. Physica-Verlag, Heidelberg 2004.

[MN06] Mendling, J., Nüttgens, M.: EPC markup language (EPML): An XML-based interchange format for event-driven process chains (EPC). Information Systems and e-Business Management 4, pp. 245–263. 2006.

[MNN05] Mendling, J., Neumann, G., Nüttgens, M.: Yet another event-driven process chain. In: Lecture Notes in Computer Science, pp. 428. 2005.

[NM06] Nickerson, J. V, Muehlen, M. zur.: The Ecology of Standards Processes: Insights from Internet Standard Making. MIS Quarterly 30, pp. 467–488. 2006.

[NR02] Nüttgens, M., Rump, F.J.: Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen P-21, pp. 64–77. 2002.

[NZ98] Nüttgens, M., Zimmermann, V.: Geschäftsprozeßmodellierung mit der objektorientierten Ereignisgesteuerten Prozeßkette (oEPK). In: M. Maicher, H.-J. Scheruhn (eds.) Informationsmodellierung - Referenzmodelle und Werkzeuge. pp. 23–35. Gabler, Wiesbaden, Germany 1998.

[Om11] OMG.: Business Process Model and Notation (BPMN) Version 2.0. 2011. Available at: http://www.omg.org/spec/BPMN/2.0/, accessed 27. Sep 2015.

[Pr95] Priemer, J.: Entscheidungen über die Einsetzbarkeit von Software anhand formaler Modelle. Pro-Universitate-Verlag, Sinzheim 1995.

[RA07] Rosemann, M., Aalst, W.M.P. van der.: A configurable reference modelling language.

Information Systems 32, pp. 1–23. 2007.
[Re05] Recker, J.C., Rosemann, M., Aalst, W.M.P. van der, Mendling, J.: On the Syntax of Reference Model Configuration – Transforming the C-EPC into Lawful EPC Models. In: Business Process Management Workshops: BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS, pp. 60–75. Nancy, France 2005.
[Ri00] Rittgen, P.: Quo vadis EPK in ARIS? Ansätze zu syntaktischen Erweiterungen und einer formalen Semantik. Wirtschaftsinformatik 42, pp. 27–35. 2000.
[Ri16] Riehle, D.M., Jannaber, S., Karhof, A., Delfmann, P., Thomas, O., Becker, J.: Towards an EPC Standardization – A Literature Review on Exchange Formats for EPC Models. In: Proceedings of the Multikonferenz Wirtschaftsinformatik (MKWI 2016), Ilmenau, Germany 2016.
[RM05] Rosemann, M., Muehlen, M. zur.: Integrating Risks in Business Process Models. Australasian Conference on Information Systems (ACIS) , pp. 62–72. 2005.
[Ro08] Rosa, M. La, Dumas, M., Hofstede, A.H.M. ter, Mendling, J., Gottschalk, F.: Beyond Control-Flow: Extending Business Process Configuration to Roles and Objects. 5231, pp. 199–215. 2008.
[Ro96] Rosemann, M.: Komplexitätsmanagement in Prozeßmodellen. Gabler, Wiesbaden 1996.
[Ru99] Rump, F.J.: Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozeßketten. B. G. Teubner, Suttgart 1999.
[RW08] Rieke, T., Winkelmann, A.: Modellierung und Management von Risiken. Ein prozessorientierter Risikomanagement-Ansatz zur Identifikation und Behandlung von Risiken in Geschäftsprozessen. Wirtschaftsinformatik 50, pp. 346–356. 2008.
[SDL05] Sarshar, K., Dominitzki, P., Loos, P.: Einsatz von Ereignisgesteuerten Prozessketten zur Modellierung von Prozessen in der Krankenhausdomäne – Eine empirische Methodenevaluation. EPK 2005 -Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten , pp. 97–116. 2005.
[SNZ97] Scheer, A.-W., Nüttgens, M., Zimmermann, V.: Objektorientierte Ergenisgesteuerte Prozeßkette (oEPK) - Methode und Anwendung. In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 141, Universität des Saarlandes 1997.
[TD06] Thomas, O., Dollmann, T.: Attributierung und Regelintegration. Proceedings of the 5th GI Workshop on Event-Driven Process Chains (EPK 2006) , pp. 49–68. 2006.
[TD08] Thomas, O., Dollmann, T.: Towards the Interchange of Fuzzy-EPCs: An XML-based Approach for Fuzzy Business Process Engineering. In: Multikonferenz Wirtschaftsinformatik, pp. 1999–2010. Garching, Germany 2008.
[TF06] Thomas, O., Fellmann, M.: Semantische Ereignisgesteuerte Prozessketten. Data Warehousing , pp. 205–224. 2006.
[Th09] Thomas, O.: Fuzzy Process Engineering. Gabler Verlag, Wiesbaden 2009.
[THA02] Thomas, O., Hüsselmann, C., Adam, O.: Fuzzy-Ereignisgesteuerte Prozessketten - Geschäftsprozessmodellierung unter Berücksichtigung unscharfer Daten. EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens , pp. 7–16. 2002.
[WS06] Winter, A., Simon, C.: Using GXL for exchanging business process models. Information Systems and e-Business Management 4, pp. 285–307. 2006.
[WW02] Webster, J., Watson, R.: Analyzing the past to prepare for the future: writing a literature review. Management Information Systems Quarterly 26, pp. 8–13. 2002.
[ZR96] Zukunft, O., Rump, F.: From Business Process Modelling to Workflow Management - An Integrated Approach. In: B. Scholz-Reiter, E. Stickel (eds.) Business Process Modelling. pp. 3–22. Springer, Berlin 1996.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 77

On the de-facto Standard of Event-driven Process Chains: Reviewing EPC Implementations in Process Modelling Tools

Arne Karhof[1], Sven Jannaber[1], Dennis M. Riehle[2], Oliver Thomas[1], Patrick Delfmann[2], Jörg Becker[3]

Abstract: Nowadays, most process modelling tools implement popular modelling languages such as the Business Process Model and Notation (BPMN) or the Event-driven Process Chain (EPC). However, in contrast to BPMN, no effort has yet been undertaken to standardize the EPC language, thus rendering EPCs as being merely a de-facto standard for business process modelling. Subsequently, this paper addresses this issue by laying ground for a successful EPC standardization. To achieve this task, several process modelling tools have been evaluated regarding their implementation of the EPC language with the objective to derive consensus about important language constructs. The evaluation reveals that there is a high degree of variety in the way tools implement EPCs. Especially syntax, semantic and pragmatic of the EPC language are not perceived homogenously and, in fact, commonly neglected. Hence, our research provides valuable implications for further EPC standardization by highlighting the state-of-the-art of the EPC from a software point of view.

Keywords: Event-driven Process Chain, EPC, Business Process Management, BPM, standardization, modelling tools, tool evaluation

1 The need for an EPC standard - How reviewing BPM tools might help

The modelling of business processes is an integral part of Business Process Management (BPM) [BNT10]. For any successful modelling endeavor, the emphasis has to be put on the choice of modelling software and business process modelling languages (BPML) [WAV04]. Studies reveal that the market volume of BPM software is continuing to grow, reaching 2.7 billion $ in 2015 [Ga15]. In 2013, over 52 vendors for BPM software compete in the broader BPM market [RM13]. Additionally, there exist several niche products of local software companies [DKK14], resulting in a large variety of potential tools to choose from. Beside proprietary notations, most tools support standardized languages such as the

[1] University of Osnabrück, Institute for Information Management and Information Systems, Katharinenstraße 3, 49069, Osnabrück, arne.karhof@uni-osnabrueck.de, sven.jannaber@uni-osnabrueck.de, oliver.thomas@uni-osnabrueck.de
[2] University of Koblenz-Landau, Institute for Information Systems Research, Universitätsstraße 1, 56016, Koblenz, riehle@uni-koblenz.de, delfmann@uni-koblenz.de
[3] University of Münster, European Research Center for Information Systems, Leonardo Campus 3, 48149, Münster, becker@ercis.uni-muenster.de

Business Process Model and Notation (BPMN) or the Unified Modeling Language (UML) for process design [DKK14]. Although the event-driven process chain (EPC) has been one of the most dominant languages for business process modelling in research and practice over the last decades [Fe09, HFL09, KS08], no systematic standardization efforts have taken place. Hence, the EPC is still considered merely a de-facto standard for business process modelling [Fe13, Wa13], though it consists of a variety of different variants [e.g. Ri16a]. The absence of an international accepted standard yields significant drawbacks for the EPC language, since the focus shifts towards standardized languages such as BPMN [DKK14]. This is primarily due to difficulties in terms of interoperability, further development and overall acceptance of modelling languages that are not based on an agreed-upon ground and thus may be implemented differently across modelling tools [Fe13].

In literature, many studies have addressed the amount of BPM software by providing an overview over the (German) BPM software market [BFV07, BS01], evaluating process modelling tools [NS02] or comparing implemented BPML [DGS10]. However, a specific focus on different EPC implementations in common BPM software has not yet been conducted. Furthermore, despite single attempts to provide specifications for parts of the EPC [e.g. MA07, NR02, Ro08], even literature has failed to reach agreement on fundamental EPC constructs. As a consequence of the combination of fragmented EPC research and arbitrary EPC implementations in software there exists no common ground in both theory and practice regarding the EPC language and its integral constructs. This, of course, hampers a systematic standard-making process guided by a standard development organization (SDO), since standardization is fundamentally based on agreement and consensus of a domain community [DG90, FKL03].

In order to prepare for a successful EPC standardization, this paper faces the stated challenges by bringing EPC implementations into perspective. Several BPM tools from both international as well as local vendors are examined in terms of degree and shape of EPC support. Hence, the paper specifically aims at providing an overview of EPC constructs as implemented in common business process modelling tools. Furthermore, the synthesis of the findings is proposed as a basis for further standardization efforts. The evaluation of various EPC implementations yields valuable insight into EPC language constructs that software companies have considered to be relevant. Therefore, the tool evaluation is able to highlight differences as well as congruities of EPC language constructs from a software point of view, which result in a proposition of constructs for standardization purposes.

The structure of the paper is organized as follows. Section 2 covers theoretical background on BPM with special focus on EPC modelling. The research methodology as applied in this paper is presented in Section 3. Subsequently, Section 4 provides an overview of identified EPC implementations. In Section 5, implications for EPC standard-making are carried out based on these findings. The paper concludes with a summary of the gained insights and an outlook on further work.

2 Process Modelling with Event-Driven Process Chains

In all kind of organizations, sets of activities are performed to achieve a superordinate business goal. Such activities, together with the organizational and technical environment they are performed in, are called business processes [We12, p.5]. BPM is about "concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes" [We12, p.5]. A key part of BPM is the creation of conceptual models to represent business processes, so called business process models.

Process models may be created using different BPML, which, for instance, can be UML, Petri nets, EPC or BPMN (see e.g. [DAH05]). One dominant language over the last decades is the EPC, which has initially been introduced at the University of Saarbrücken by [KNS92] in 1992. EPC models consist of alternating events and activities, which represent the process flow. Connectors can be used to split or merge the process flow as needed. For this reason, [KNS92] defines the AND connector (all subsequent process flows are performed), the XOR connector (exactly one process flow is performed) and the OR connector (one or more process flows are performed). Later, organizational units, information systems and information objects were added to EPC models to enrich functions with further details [HKS93].

Over the years, the EPC language has been further developed, extended and modified by a large number of different authors. For example, [Pr95] defines a sequence connector (where several process flows are performed in a sequence of any order), [Ro96] presents a decision table connector (ET) and the OR1 connector. New elements have been added to the EPC to model business risks [BO02, RM05, RW08] or to model fuzzy decision-making [TD06, THA02]. New relationships have been added to not only include the process-flow, but also the data-flow in EPC models [ZR96].

Other authors have added different concepts to EPC models, such as [HWS93], who link EPC models to a state-machine for real-time execution, or [TF06] and [FKS09], who annotate EPC models with concepts from an ontology. For different use cases of the EPC, there are several different and incompatible file-based exchange formats [Ri16b]. The idea of EPCs has also been transferred to different application scenarios, for example object-oriented business process modelling [NZ98, SNZ97], modelling of state-based workflow patterns [MNN05] or service-based process modelling [HW07]. Lastly, EPC has also been applied to configurative reference modelling [RA07].

In order to support design and administration of business processes, there are several commercial and a few non-commercial solutions for managing process models. Several studies provide an overview of the market for BPM tools [BH14, DKK14, Ga15, RM13]. As the number of BPM tools on the market is large, different variations and adoptions of the EPC are manifold. As there is no established standard for EPC, it is obvious that the implementation of EPC differs between BPM tools.

3 A methodology for identifying and evaluating BPM tools

In order to provide a comprehensive and holistic overview of the BPM software market, a structured procedure model is applied. For this purpose, we draw upon methodologies coming from marketing science and transform these methods to fit our needs. In [CM99], a systematic procedure model is introduced to support managers in the assessment of their market for potential competitors. A similar approach is presented by [BP02], who use a two-step framework for competitor identification and analysis. In essence, both models share the way in which an unstructured and crowded environment is grasped, and objects of interest are identified and evaluated. We apply this general process to the situation at hand. In this case, the environment is represented by the BPM software market. The detailed procedure model followed throughout this paper is depicted in Figure 1.

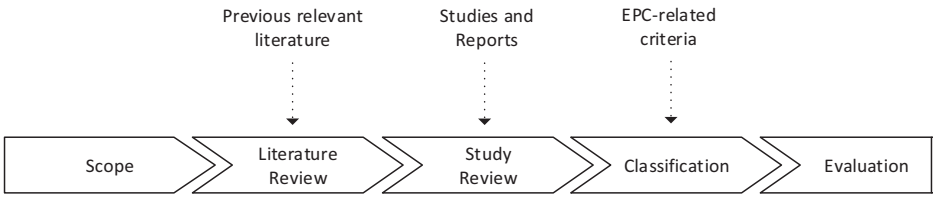


Fig. 1: Applied methodology

Using a five-step process, BPM tools are identified and analyzed. Regarding the scope, we limited our search to international software vendors who explicitly offer BPM software. Hence, workflow modelling suites have not been considered. Additionally, software tools had to be available for testing purposes. Therefore, we excluded vendors that did not provide any form of test or demo access to their software. Similarly, tools that could not have been installed due to technical circumstances have been omitted. Lastly, the final number of BPM tools has been shortened according to their support of the EPC language. Following the setting of the scope, relevant literature has been reviewed. This includes, for example, scientific contributions regarding EPC language constructs, but also related research-driven evaluations of BPM tools. Next, insight into the BPM market has been gained by taking various studies and market reports coming from institutions such as Gartner [SH10], the Fraunhofer Institute [DKK14], Forrester Research [RM13] or Ovum [BH14] into account. The final list has been analyzed using predefined criteria, which directly facilitate the identification of similarities and differences of EPC implementations in BPM tools. The criteria applied in this paper are presented in Table 1. For all EPC implementations, the state of EPC syntax, semantic and pragmatic is investigated. In doing so, we refer to respective sets of rules carried out in [Fe13]. Exemplarily, “1” as a characteristic of the syntax criterion indicates that a particular EPC implementation adheres to syntax rule “1” as presented in [Fe13]. Reviewing the state of syntactical, semantical and pragmatical rules covered by EPC implementations facilitates the understanding of what aspects of the EPC language are considered important in practice,

hence supporting the determination of a standardized EPC specification that meets practical demands.

Criterion	Example	Description
Syntax	1,2,5	Refers to the set of syntactical rules as presented in [Fe13]
Semantic	3,4	Refers to the set of semantic rules as presented in [Fe13]
Pragmatic	1,6	Refers to the set of pragmatic rules as presented in [Fe13]
Elements	●	Does the extension introduce new elements beyond [Ro96]?
Connectors	●	Does the extension introduce new connector types beyond [Ro96]?
Checking	M	(M)anual or (D)esign time syntax check
Exchange format	XML	Type of EPC exchange format provided
Guidelines	Tooltips	Guidelines that support the modeler in the creation of sound EPC models

Tab. 1: EPC implementation evaluation criteria

Besides syntax, semantic and pragmatic, we expand the evaluation to modelling guidance and the extent of language constructs covered as well as their graphical representation. For this purpose, we use EPC elements and connector types summarized in [Ro96] as a baseline for further investigation. In particular, it is of interest whether an EPC implementation adheres to these elements resp. connector types. Based on potential similarities or deviations in terms of elements introduced or their graphical layout, a consensus of these elements can be reached from a software perspective. The Checking criterion is used to assess if the tool implements a manual or design-time based checking of the syntax, semantic and pragmatic of an EPC model, which provides insight regarding the importance of model soundness as perceived from BPM software vendors. Furthermore, having a look at the implemented exchange format is used to formulate a consensus regarding a standard exchange format for EPC models. Lastly, it is subject of analysis whether and how EPC implementations support the modeler in creating sound and meaningful EPC models.

Finally, the evaluation of tools using the aforementioned criteria is used to investigate the state of the EPC modelling language from a software point of view, hence to a certain degree reflecting needs that come from practice. The evaluation step not only provides an overview of a tool's capability to handle EPC modelling, but also gives additional input whether to include EPC language constructs in a potential EPC standard that have proven to be applicable and beneficial in practice.

4 An overview of EPC implementations in BPM tools

After setting the scope of our analysis, considering previous relevant literature and integrating similar studies about BPM software (cf. Fig. 1), we initially identified a set of 78 contemplable solutions. At first, no software has been excluded from our investigation. The list included leaders like Pegasystems or Appian, challengers, e.g. Fujitsu, niche players, exemplary Newgen or visionaries such as Intalio or BizAgi [SH10]. While classifying the first set according to our specified criteria in Section 3, we were able to remove 64 software solutions in total that did not fit our requirements. Thus, only 14 software providers have been identified that support EPCs and offer a free trial. We analyzed these solutions according to the predefined criteria. The results are presented in Table 1. Due to lack of space, we relinquish and outline the presentation of some categories. Overall, we relinquish the category miscellaneous, where we recorded additional elements that are not directly related with business process modelling or information about the underlying software like Eclipse or Microsoft Visio. Also, we spare the detailed enumeration of all elements the solutions provide, as we pit the extent against the native EPC components.

First, we recorded the supported business process languages to solely include EPC-supporting BPM solutions. Since we specifically examine EPC functionalities, all of the listed software can be considered as EPC compatible. During the examination, we checked if not only the standard EPC [KNS92] is supported, but also if any EPC extensions can be modeled. Except for *EPC Tools*, every solution supports the extended EPC [HKS93]. The *Bflow** platform additionally provides modelling with object-oriented EPCs [SNZ97]. Multiple tools also offer additional elements, but did not implement a specific EPC extension known from literature [e.g. SDL05]. We fastidious listed every additional element, adjusted the degree of abstraction and used circles as the form of representation. Thereby, a filled circle in the column “Elements” declares that the tool not only covers the extended EPC elements but also additional, partly tool-specific, elements. The software *EPC Tools* only supports the plain EPC and therefore is the only software figured as a bar. Unfilled circles represent just eEPC modelling elements support. Accordingly, unfilled circles in the column “Connectors” signify the AND, OR, XOR operators and the control/information flow. The only software solution that offers more than these basic constructs is the *Bflow** platform that additionally supports relation flows between EPC elements.

Product	Elements	Connectors	Syntax	Semantic	Pragmatic	Checking	Exchange formats	Guidelines
Cockpit Designer	●	○	2 7	—	2	D	—	Recommender
Aris Express	●	○	2 3	—	—	D	—	Ref. Models
Bflow*	●	●	1 2 4 6 7	●	1 2	D	XMI	Tooltips
BIC Platform	●	○	3	—	—	—	XML	Recommender
Cubetto	●	○	—	—	—	—	XML, CSV	—
Edraw Max	●	○	—	—	—	—	XML	—
EPC Tools	—	○	1 6	—	—	M	EPML	—
iGrafx	●	○	—	—	—	M	—	—
Process Modeler for Visio	○	○	—	—	—	—	BPEL, XPDL	—
SemTalk	●	○	1 2 3 5 6 7	—	2 6	D	XML	—
Signavio Process Editor	●	○	1 2 3 5 6 7	—	2 10	M	XML, XPDL	—
Symbio Modeling Client	●	○	2 6	—	1 2	D	—	—
ViFlow	●	○	—	—	—	—	—	—
Visio	○	○	—	—	—	—	XML	—

Tab. 2: Evaluation of EPC supporting BPM Software

The examination of quality criteria revealed an outlier for the second syntactic criterion, which requires a continual transition between events and functions. Many tools do not strictly demand the adherence to this rule, since they permit the omission of (trivial) events. In case this broad interpretation of the second rule has been implemented, the corresponding rule “2” in Table 1 has been underlined. Another striking point is the filled circle in the row of the *Bflow** tool. Because no tool supported any semantically validation, the *Bflow** tool stood out as being the only software that at least demands that the process model entities have to be labeled. Although this is no criterion stated in [Fe13], we decided to document it in the table nevertheless. The last two categories illustrate if there is any exchange format for EPC models supported by the solution and if there is any kind of guidance for the user while modelling in EPC. As the table constitutes, multiple solutions provide the import and export of specific exchange formats. However, the specific type of exchange format heavily differs. Although just three entries could be done in the last category, a heterogeneous distribution can be observed. The three addressed solutions pursue different approaches as there are either tooltips that pop up during design-time, reference models that the user can use as templates or a recommendation system that supports the user in deciding what element to model next.

5 Implications for EPC standardization

As our findings indicatethere is hardly a consensus between software vendors in the field of BPM regarding essential EPC constructs. For our next step, it has to be discussed to what degree the insights gained from reviewing BPM tools influence the proposed standardization process. In order to solve this debate, stakeholders of a standard need to be considered. Possible stakeholders are consultants, academia, research institutes, and governmental agencies [LK06]. From a high-level point of view, these groups can either be considered as (academic) researchers or practical applicants, who do not necessarily have to be disjoint groups. However, for the sake of argument we assume that these groups have different roles regarding the impact factor on standardization. On the one hand, research interest groups are more likely to take part in the (continued) development process of a standard, while on the other hand practical applicants fulfill their role by implementing and actual using this standard. Hence, we argue that the developer point of view of a business process modeling language best reflects the acceptance or denial of specific language constructs in practice. Therefore, it is possible to derive insights about whichcomponents to consider in an EPC standardization process and which not. Furthermore, as Table 1 already demonstates, our examination highlights the negative outcome a missing standard can cause. The practical implementation of EPC modelling is a strikingly heterogonous area, as the software vendors were never able to adhere to an agreed-upon basis for the EPC language.

In favor of a better overview of this diversified field of application, we analyzed our results as shown in Figure 1. Especially the overlaps, where the software vendors achieved agreement among themselves, are of particular interest, as derivations for an EPC standard

can directly be deduced. Unfortunately, as depicted in Figure 1, the commonalities are highly underrepresented. Regarding the elements category, only 21.4% still support solely the EPC and eEPC basis elements. However, it is worth mentioning that 78.6% does not imply that all of the vendors actually use *one* set of EPC elements but *their own* notation set that exceeds the eEPC (illustrated by "eEPC+"). Therefore, the majority of EPC tools extends the (e)EPC with additional, self-developed elements. The second category is the only one where a major consensus could be identified. Only 7.1% of all vendors use additional, own developed connector types. Consequently, from a practical point of view, it seems like the standard EPC connector types already cover most of the users' requirements. The following three categories illustrate the distribution of quality checking in regard to syntax, semantic and pragmatic. While syntax is the only category which is supported by more than half of the tools, the nearly non-existing semantic validation stands out. However, the pragmatic quality seem to be well-covered by BPM tools. Furthermore, as the binary classification only represents the number of tools supporting quality checking, but not *the number of criteria* covered, we added an additional value for the average support. This value is calculated as follows:

$$\emptyset - \text{Support} = \frac{\sum_{i=1}^{N} \frac{T_i}{|T|}}{N}$$

The variables are as follows: N = quantity of quality criteria based on [Fe13], i = the specific quality criterion, and T_i = number of tools who support the specific quality criterion i. $|T|$ indicates the cardinality of the set T. This value is equal to the overall number of evaluated tools. In our case, this value always signifies 14. Exemplary, the value for average support of the pragmatic quality can be obtained by calculating:

$$\emptyset - Support_{Prag.} = \frac{\frac{2}{14} + \frac{6}{14} + \frac{0}{14} + \frac{0}{14} + \frac{0}{14} + \frac{1}{14} + \frac{0}{14} + \frac{0}{14} + \frac{0}{14} + \frac{1}{14}}{10}$$

$$\emptyset - Support_{Prag.} = \frac{0.714286}{10} = 0.07143 \triangleq 7.1\%$$

Considering these values of the three quality categories (illustrated in Figure 1 as hatched areas), it is evident that every category is insufficiently supported by BPM tools. Even the syntactical category, which demands criteria relatively easy to implement, is only associated with an average support of 22.3%. Furthermore, the average semantic support is nearly nonexistent (0.4%). The pragmatic support is weakly represented as well. Finally, the last three categories present the statistically analysis of checking support, provided exchange formats and user guidance. While most of the tools support some form of verification or validation while modelling, it has to be kept in mind that the corresponding Ø-support values show that this support is heavily constrained. Additionally, most of the tools provide some sort of exchange format. However, as Figure 1 depicts, besides using XML as an underlying structure, software vendors have not been able to reach an agreement on an exchange standard. The same problem can be observed with user

supporting guidelines. While only three software solutions (21.4%) provide any kind of user guidance, they all confine to different approaches.

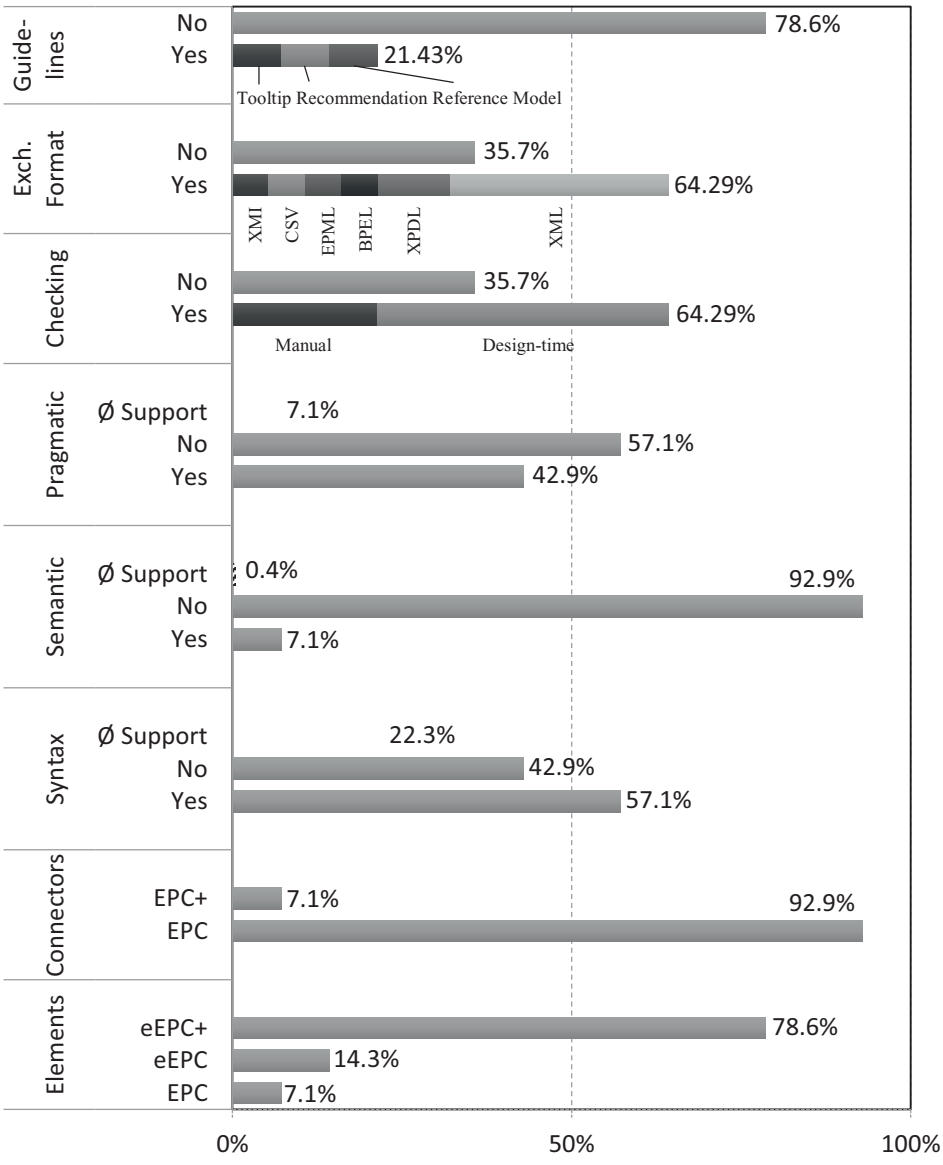


Fig. 2: Evaluation of BPM-Tools

Following the elicitation and evaluation of BPM-Tools, it is of interest to draw - in the best case obvious - implications towards EPC standardization. Our initial intension to interpret commonalities is hampered by the fact that, besides the category Connectors, there are no distinct overlaps among the software solutions. Despite this circumstance, we can ascertain the following points regarding to our evaluation categories:

Elements: Based on the percentage distribution, we assume that the basic (e)EPC elements seem not to completely fit the needs of nowadays users. The majority of software vendors use additional elements, partly self-developed, partly from other BPM languages or proposals from academic research. Nevertheless, it could be observed that most of the tools did use a similar layout for their elements. This includes the hexagon for events, the rectangle for functions (mostly with rounded edges) and circular connectors. However, the labelling of connectors is again diversified in EPC implementation (XOR vs. X). For future work, it will become necessary to value those additional elements in favor of deciding whether to integrate them in an EPC standard.

Connectors: As nearly all solutions use the basis set of EPC connectors, the general applicability can be underlined. Self-evidently, the outlier has to be examined in terms of determining its usability.

Syntax, Semantic, Pragmatic: In this area, there is a heavy mismatch among the tools. It can be concluded that the EPC would heavily benefit from an underlying and formal defined standard as a basis for EPC implementations. Unfortunately, as there are almost no commonalities, explicit rules for EPC standardization cannot be deduced. A single implication that can be made is that the difficulty of verifying EPC quality seems to vary in regard to the respective category. Based on the findings we assume that the semantic quality is the most challenging task for computer-based verification.

Checking: Since the evaluation presents roughly the same distribution between design-time and manual-triggered checking of syntactical, semantical or pragmatically aspects, an evident recommendation from practice cannot be made. Additionally, no coherence between the type of checking and the nature and extent of covered rules could be detected. Generally, a design-time approach seems to be more applicable from a practical perspective, as it prevents users from working with erroneous models. However, it has to be kept in mind that not every error detected by a checking mechanism necessarily has to be an actual error of the underlying process model. Hence, the checking mechanism must not be too restrictive for practical application. In general, the review of checking mechanisms implies that, considering a potential EPC standard, emphasis has to be put on the degree of complexity and restrictions in order to meet practical demands.

Exchange Formats: The area of exchange formats represents the most diversification. Out of nine software vendors, who offer any type of exchange format, six different approaches are pursued. The most common format is XML. Since many other listed formats are also based on XML, namely XMI, EPML, BPEL and XPDL, it can be seen as the major choice for data communication between EPC tools. This heavily implies that a XML-based exchange format might be feasible for an EPC standard.

Guidelines: The provided user guidance is rather underrepresented. Any concrete implications for an EPC standard cannot be deduced. In future, the usability of proposed guidance types has to be valued. Furthermore, it needs to be clarified if the presence of user guidance is actually necessary if there is already a checking mechanism.

6 Conclusion

In order to prepare for EPC standardization, the similarities and differences between EPC implementations in common BPM software has to be revealed. Subsequently, we identified BPM tools that implement the EPC modelling language and evaluated them against our predefined criteria. Altogether, we analyzed 14 tools measured against 8 categories and 34 different quality criteria.. Our findings show that the negative effects caused by the abstinence of a universally accepted EPC standard are prevalent. Regarding the predefined criteria, there has been no consensus in terms of EPC language constructs. The only commonalities that could be identified regard the basic layout of EPC elements and EPC connectors. Most notably, the diversification of quality criteria regarding the syntax, semantic and pragmatic is apparent. Despite 64% of identified tools provide a basic checking mechanism, most of them only consider a small subset of rules. The majority of tools leave the creation of valid process models solely to the user. This may imply a lack of importance for semantic, pragmatic and syntactic issues in practice.

By evaluating, we could further deduce implication for standardization purposes and gain valuable insights about the practical implementation of the EPC. Explicit conclusions could be made regarding the layout, the set of connectors, and the exchange format. In terms of the layout, all EPC implementations share the same graphical representation. Accordingly, also the set of connectors is identical among the tools. Therefore, we conclude that there is an agreement regarding the layout of EPC elements that has to be considered for EPC standard making. Furthermore, we revealed that exchange formats for EPC modelling are dominated by XML. Hence, we suggest a XML-based EPC exchange format to be included in a future EPC standard.

Based on our research, we are able to state and underline the urgent need for an EPC standard. In this paper, we evaluated EPC modelling from a practical point of view and uncovered the consequences of a non-standardized process modelling language. In conclusion, our research contributes to the body of knowledge in two ways. First, we shed light on the BPM software market and provide an overview of core players in EPC modelling which supports both academic and practical disciplines, as we build groundwork for further research and application in the field of EPC modelling. Second, we investigated EPC implementations in order to achieve a consensus regarding essential EPC language constructs from a software point of view.

Acknowledgement: The research presented in this paper is part of the SPEAK project and is funded by the Federal Ministry for Economic Affairs and Energy (BMWi) under grant number 01FS14030.

References

[BFV07] Bartels, G., Frank, P., Völz, M.: Vergleich von BPMN-Modellierungswerkzeugen. Stuttgart 2007.

[BH14] Barnett, A.G., Holt, M.: Ovum Decision Matrix : Selecting a Business Process Management Solution , 2014. 2014.

[BNT10] Becker, J., Niehaves, B., Thome, I.: How Many Methods Do We Need ? – A Multiple Case Study Exploration into the Use of Business Process Modeling Methods in Industry. In: AMCIS 2010 Proceedings, 2010.

[BO02] Brabänder, E., Ochs, H.: Analyse und Gestaltung prozessorientierter Risikomanagementsysteme mit Ereignisgesteuerten Prozessketten. In: EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens, pp. 1–5. 2002.

[BP02] Bergen, M., Peteraf, M. a.: Competitor Identification and Competitor Analysis: A Broad-Based Managerial Approach. Managerial and Decision Economics 23, pp. 157–169. 2002.

[BS01] Bullinger, H.-J., Schreiner, P.: Business Process Management Tools: Eine evaluierende Marktstudie über aktuelle Werkzeuge. Stuttgart 2001.

[CM99] Clark, B.H., Montgomery, D.B.: Managerial Identification of Competitors. Journal of Marketing 63, pp. 67–83. 1999.

[DAH05] Dumas, M., Aalst, W.M.P. van der, Hofstede, A.H.M. ter.: Process-aware Information Systems - Bridging People and Software through Process Technology. John Wiley & Sons, Hoboken, New Jersey 2005.

[DG90] David, P.A., Greenstein, S.: The Economics of Compatibility of Standards: An Introduction to Recent Research. Economics of Innovation and New Technology 1, pp. 3–41. 1990.

[DGS10] Drawehn, J., Gayer, S., Schneider, P.: Business Process Modeling 2010: Modellierung von ausführbaren Geschäftsprozessen mit der Business Process Modeling Notation. Fraunhofer, Stuttgart, Germany 2010.

[DKK14] Drawehn, J., Kochanowski, M., Kötter, F.: Business Process Management Tools 2014. Fraunhofer, Stuttgart, Germany 2014.

[Fe09] Fettke, P.: Ansätze der Informationsmodellierung und ihre betriebswirtschaftliche Bedeutung: Eine Untersuchung der Modellierungspraxis in Deutschland. Schmalenbachs Zeitschrift für betriebswirtschaftliche Forschung (zfbf) 61, pp. 550–580. 2009.

[Fe13] Fellmann, M., Bittmann, S., Karhof, A., Stolze, C., Thomas, O.: Do We Need a Standard for EPC Modelling? The State of Syntactic, Semantic and Pragmatic Quality. In: 5th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA), pp. 103–116. St. Gallen, Switzerland 2013.

[FKL03] Fomin, V., Keil, T., Lyytinen, K.: Theorizing about standardization: integrating fragments of process theory in light of telecommunication standardization wars. Sprouts: Working Papers on Information Environments, Systems and Organizations 3, pp. 29–60. 2003.

[FKS09] Filipowska, A., Kaczmarek, M., Stein, S.: Semantically annotated EPC within

semantic business process management. In: Lecture Notes in Business Information Processing, pp. 486–497. Milano, Italy 2009.

[Ga15] Gartner Inc.: Gartner Says Spending on Business Process Management Suites to Reach $2.7 Billion in 2015 as Organizations Digitalize Processes. 2015. Available at: http://www.gartner.com/newsroom/id/3064717, accessed 15. Oct 2015.

[HFL09] Houy, C., Fettke, P., Loos, P.: Stilisierte Fakten der Ereignisgesteuerten Prozesskette – Anwendung einer Methode zur Theoriebildung in der Wirtschaftsinformatik. In: EPK 2009. 8. Workshop der Gesellschaft für Informatik e.V. (GI) und Treffen ihres Arbeitkreises "Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK). Gesellschaft für Informatik, pp. 22–41. Bonn, Germany 2009.

[HKS93] Hoffmann, W., Kirsch, J., Scheer, A.-W.: Modellierung mit Ereignisgesteuerten Prozeßketten. In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 101, Universität des Saarlandes 1993.

[HW07] Huth, S., Wieland, T.: Geschäftsprozessmodellierung mittels Software-Services auf Basis der EPK. In: V. Nissen, M. Petsch, H. Schorcht (eds.) Service-orientierte Architekturen. pp. 61–76. Deutscher Universitäts-Verlag, Wiesbaden 2007.

[HWS93] Hoffmann, W., Wein, R., Scheer, A.-W.: Konzeption eines Steuerungsmodells für Informationssysteme – Basis für die Real-Time-Erweiterung der EPK (rEPK). In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 106, Universität des Saarlandes 1993.

[KNS92] Keller, G., Nüttgens, M., Scheer, A.-W.: Semantische Prozeßmodellierung auf der Grundlage “Ereignisgesteuerter Prozeßketten (EPK).” In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 89, Universität des Saarlandes 1992.

[KS08] Knuppertz, T., Schnägelberger, S.: Status Quo Prozessmanagement 2007/2008 – Ergebniszusammenfassung. Komptenzzentrum für Prozessmanagement 2008.

[LK06] Lyytinen, K., King, J.L.: Standard Making: A Critical Research Frontier for Information Systems Research. MIS Quarterly 30, pp. 405–411. 2006.

[MA07] Mendling, J., Aalst, W.M.P. van der.: Formalization and Verification of EPCs with OR-Joins Based on State and Context. Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE’07) 4495, pp. 439–453. 2007.

[MNN05] Mendling, J., Neumann, G., Nüttgens, M.: Yet another event-driven process chain. In: Lecture Notes in Computer Science, pp. 428. 2005.

[NR02] Nüttgens, M., Rump, F.J.: Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen P-21, pp. 64–77. 2002.

[NS02] Nägele, R., Schreiner, P.: Bewertung von Werkzeugen für das Management von Geschäftsprozessen. Zeitschrift Führung und Organisation 71, pp. 201–210. 2002.

[NZ98] Nüttgens, M., Zimmermann, V.: Geschäftsprozeßmodellierung mit der objektorientierten Ereignisgesteuerten Prozeßkette (oEPK). In: M. Maicher, H.-J. Scheruhn (eds.) Informationsmodellierung - Referenzmodelle und Werkzeuge. pp. 23–35. Gabler, Wiesbaden, Germany 1998.

[Pr95] Priemer, J.: Entscheidungen über die Einsetzbarkeit von Software anhand formaler Modelle. Pro-Universitate-Verlag, Sinzheim 1995.

[RA07] Rosemann, M., Aalst, W.M.P. van der.: A configurable reference modelling language. Information Systems 32, pp. 1–23. 2007.

[Ri16a] Riehle, D.M., Jannaber, S., Karhof, A., Thomas, O., Delfmann, P., Becker, J.: On the de-facto Standard of Event-driven Process Chains: How EPC is defined in Literature. In: Proceedings of the Modellierung 2016, Karlsruhe, Germany 2016.

[Ri16b] Riehle, D.M., Jannaber, S., Karhof, A., Delfmann, P., Thomas, O., Becker, J.: Towards an EPC Standardization – A Literature Review on Exchange Formats for EPC Models. In: Proceedings of the Multikonferenz Wirtschaftsinformatik (MKWI 2016), Ilmenau, Germany 2016.

[RM05] Rosemann, M., Muehlen, M. zur.: Integrating Risks in Business Process Models. Australasian Conference on Information Systems (ACIS) , pp. 62–72. 2005.

[RM13] Richardson, C., Miers, D.: The Forrester Wave™: BPM Suites, Q1 2013. 2013.

[Ro08] Rosa, M. La, Dumas, M., Hofstede, A.H.M. ter, Mendling, J., Gottschalk, F.: Beyond Control-Flow: Extending Business Process Configuration to Roles and Objects. 5231, pp. 199–215. 2008.

[Ro96] Rosemann, M.: Komplexitätsmanagement in Prozeßmodellen. Gabler, Wiesbaden 1996.

[RW08] Rieke, T., Winkelmann, A.: Modellierung und Management von Risiken. Ein prozessorientierter Risikomanagement-Ansatz zur Identifikation und Behandlung von Risiken in Geschäftsprozessen. Wirtschaftsinformatik 50, pp. 346–356. 2008.

[SDL05] Sarshar, K., Dominitzki, P., Loos, P.: Einsatz von Ereignisgesteuerten Prozessketten zur Modellierung von Prozessen in der Krankenhausdomäne – Eine empirische Methodenevaluation. EPK 2005 -Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, pp. 97–116. 2005.

[SH10] Sinur, J., Hill, J.B.: Magic Quadrant for Business Process Management Suites. 2010.

[SNZ97] Scheer, A.-W., Nüttgens, M., Zimmermann, V.: Objektorientierte Ergenisgesteuerte Prozeßkette (oEPK) - Methode und Anwendung. In: Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), No. 141, Universität des Saarlandes 1997.

[TD06] Thomas, O., Dollmann, T.: Attributierung und Regelintegration. Proceedings of the 5th GI Workshop on Event-Driven Process Chains (EPK 2006) , pp. 49–68. 2006.

[TF06] Thomas, O., Fellmann, M.: Semantische Ereignisgesteuerte Prozessketten. Data Warehousing , pp. 205–224. 2006.

[THA02] Thomas, O., Hüsselmann, C., Adam, O.: Fuzzy-Ereignisgesteuerte Prozessketten - Geschäftsprozessmodellierung unter Berücksichtigung unscharfer Daten. EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens, pp. 7–16. 2002.

[Wa13] Walterbusch, M., Grove, S., Breitschwerdt, R., Stolze, C., Teuteberg, F., Thomas, O.: Case-based Selection of Business Process Modeling Tools : An Evaluation Criteria Framework. AMCIS 2013 Proceedings, pp. Enduser IS general presentation 21. 2013.

[WAV04] Weske, M., Aalst, W.M.P. van der, Verbeek, H.M.W.: Advances in business process management. Data & Knowledge Engineering 50, pp. 1–8. 2004.

[We12] Weske, M.: Business Process Management. Springer, Heidelberg 2012.

[ZR96] Zukunft, O., Rump, F.: From Business Process Modelling to Workflow Management - An Integrated Approach. In: B. Scholz-Reiter, E. Stickel (eds.) Business Process Modelling. pp. 3–22. Springer, Berlin 1996.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 93

NESTML: a modeling language for spiking neurons

Dimitri Plotnikov[1], Bernhard Rumpe[1], Inga Blundell[2], Tammo Ippen[2,3], Jochen Martin Eppler[4] and Abigail Morrison[2,4,5]

Abstract:

Biological nervous systems exhibit astonishing complexity. Neuroscientists aim to capture this complexity by modeling and simulation of biological processes. Often very complex models are necessary to depict the processes, which makes it difficult to create these models. Powerful tools are thus necessary, which enable neuroscientists to express models in a comprehensive and concise way and generate efficient code for digital simulations. Several modeling languages for computational neuroscience have been proposed [Gl10, Ra11]. However, as these languages seek simulator independence they typically only support a subset of the features desired by the modeler. In this article, we present the modular and extensible domain specific language NESTML, which provides neuroscience domain concepts as first-class language constructs and supports domain experts in creating neuron models for the neural simulation tool NEST. NESTML and a set of example models are publically available on GitHub.

Keywords: Simulation, modeling, biological neural networks, neuronal modeling, neuroscience, NEST, NESTML, MontiCore, domain specific language, code generation, C++.

1 Introduction

Classical neuroscience investigates the biophysical processes behind single neuron behavior and higher brain function. The first experimental studies of the nervous system were conducted already hundreds of years ago [Sh91], but observing single-cell activity in a cell culture or slice (*in vitro*) or in an intact brain (*in vivo*) is a technically challenging task. It was thus not before the beginning of the last century that details about the structure and function of the building blocks of the brain became known.

In the early 40s of the last century, McCulloch and Pitts [MP43] explored the idea of using simple threshold elements to mimic the behavior of interconnected nerve cells. However, it soon turned out that these artificially built circuits were too simple and limited to study the principles at work in living brains.

[1] RWTH Aachen University, Chair of Software Engineering, Jülich Aachen Research Alliance (JARA), Ahornstraße 55, 52074 Aachen, Germany
[2] Forschungszentrum Jülich, Institute of Neuroscience and Medicine (INM-6), Institute for Advanced Simulation (IAS-6), JARA BRAIN Institute I, 52025 Jülich, Germany
[3] Norwegian University of Life Sciences, Dept. of Mathematical Sciences and Technology, 1432 Ås, Norway
[4] Forschungszentrum Jülich, Simulation Lab Neuroscience, Bernstein Facility for Simulation and Database Technology, Institute for Advanced Simulation, JARA, 52025 Jülich, Germany
[5] Ruhr-University Bochum, Faculty of Psychology, Institute of Cognitive Neuroscience, 44801 Bochum, Germany

The field of neural networks consequently split: the descendants of the early neural networks are still used under the term *artificial neural networks* (ANN) to solve learning and classification tasks in engineering applications. Biologically more plausible models of neural circuits are nowadays known as *spiking* or *biological neural networks*.

1.1 Neural modeling and simulation

Computational neuroscience builds models for nerve cells (*neurons*) and their connections (*synapses*) that capture certain aspects of their anatomy and physiology. Depending on the study, different aspects are important (Section 2). As theoreticians prefer to use the simplest model that still exhibits the behavior they are interested in, a multitude of different models was published. The level of detail ranges from *compartmental models* that include many biophysical details to reduced *point neuron models* that describe the basic quantities or the cell by a small set of differential equations (Section 2). The simulation of networks of such model neurons (i.e. the propagation of the underlying equations in time) allows to execute *in silico* experiments to test hypotheses in a stable and controllable environment.

As the simulation of different classes of neurons requires different technical infrastructure (e.g. for the storage of connections or the communication between elements), different simulators have been developed. Each of them is specialized on a specific part of the spectrum of modeling tasks. This makes it hard to develop new neuron and synapse models in a general way and even harder to compare and verify findings across simulators, since models must be re-implemented for every simulator [Cr12].

To ease model-sharing and improve reproducibility in the field, several modeling languages were conceived (Section 3). They usually consist of the language itself and tools to generate a model implementation from a model specification. As the majority of the languages are simulator agnostic, they cannot take advantage of the convenience functions of a given simulator. This often results in models with lower performance or accuracy compared to a hand-written version of the same model.

1.2 The neural simulation tool NEST

NEST [GD07] is a simulator for large networks of spiking point neurons available as open source software (www.nest-simulator.org). Using hybrid parallelization it runs on all machines from laptops to the world's largest supercomputers [He12, Ku14]. Over 450 published studies used NEST and 360 users are currently subscribed to the mailing list. Due to its reliability and popularity, NEST was selected as simulator for brain-scale networks of simplified neurons in EU's Flagship *Human Brain Project* (http://humanbrainproject.eu).

At the outset of this study, NEST contained 36 neuron models, each of which implemented by hand as a C++ class using NEST's model API and embedded into NEST's infrastructure. Developing new models requires expert knowledge of the neuroscience context, as well as of C++ and NEST's internals. Changes to NEST's infrastructure or API often require changes to all models, which impairs the maintainability of NEST.

The C++ classes mix the model description (i.e. the equations and algorithms governing the dynamic behavior) with the model implementation, which impairs model comprehension. An example are the linear models in NEST, which use an exact solution for the differential equation rather than one obtained by a general solver [RD99]. This hides the actual equations deep in the model code.

Due to the lack of modularity in the C++ model code, new neuron models are mostly created by *copy&paste* from existing models. The fact that this task is often carried out by neuroscientists who are not experts in programming leads to redundancy, suboptimal performance, improper documentation and reduced maintainability. Preliminary investigations show cases where two models share more than 90% of their implementation.

1.3 The NEST modeling language NESTML

NESTML is a domain specific language that supports the specification of neuron models in a precise and concise syntax, which is familiar to the domain experts. Model equations can either be given as a simple string of mathematical notation or as an algorithm written in the built-in procedural language. The equations are analyzed by NESTML to compute an exact solution if possible or use an appropriate numeric solver otherwise (Section 4).

The simplicity of the explicit syntax of NESTML guarantees good comprehensibility and a clear separation between the model specification and its implementation. A code generator creates optimized model code alongside auxiliary code to load the model dynamically into NEST (Section 5).

First class modularization concepts in the language simplify the reuse of neuron definitions and parts thereof. This feature fosters the re-use of well tested components in models instead of re-implementing them. Models expressed in other languages can be compiled to NESTML by the code generation tools of the language.

Being built on top of the language workbench MontiCore [KRV07, KRV08], all tools belonging to NESTML are generated from a language grammar. This allows us to conveniently update the language itself to new modeling requirements, and the code generator to changes in the NEST infrastructure or API.

2 Modeling spiking neurons

As with all body cells, neurons are also confined by a membrane. *Channels* embedded into the membrane selectively allow certain types of ions to pass, active transporter molecules move ions in and out of the cell. These mechanisms maintain up a gradient of charges, resulting in an electrical potential across the membrane.

An incoming signal (*action potential*, *spike*) leads to a short excursion of the membrane potential. The direction of the excursion depends on the type of the sending (*presynaptic*)

neuron, which can be either *excitatory* (positive excursion) or *inhibitory* (negative excursion). If the input is strong enough or several inputs occur simultaneously, the membrane potential eventually reaches a *threshold* and the neuron fires a spike itself. Spikes are transmitted via the synapses to receiving (*postsynaptic*) neurons, where the spike again leads to a change of the membrane potential. After emitting a spike, a neuron is inactive for a certain time, called its *refractory period* [Ni01].

The work of Lapicque [La07] and later Hodgkin and Huxley [HH52] paved the way for creating models of neurons with biologically realistic parameters. For the membrane potential, they define an equivalent electrical circuit, in which the membrane itself is represented by a capacitor, ion channels by resistors and external inputs by an additional current.

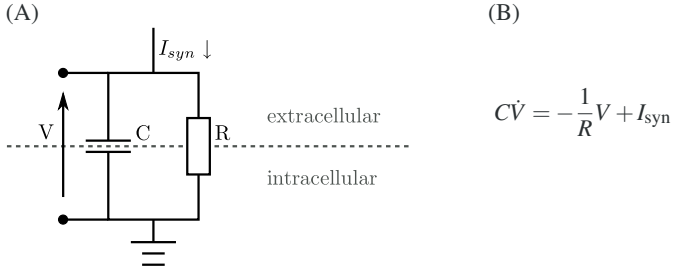


Figure 1: Electrical circuit corresponding to single compartment of a neuron. (A) circuit diagram. (B) differential equation for the membrane potential V given capacitance C, resistance R and external input current I_{syn}

.

A common approach to modeling neurons is to divide the 3D reconstruction of a real neuron into compartments and use one Hodgkin and Huxley circuit for each compartment. The compartments are coupled using the formalism of cable theory. In case the morphology of such a *multi-compartment model* only consists of a single compartment, it is called a *point neuron model*.

The basic differential equation shown in Figure 1 only characterizes the subthreshold dynamics of the neuron. Checks for threshold crossings, spike generation and refractoriness are usually added in an algorithmic fashion using conditionals and wait cycles. Spiking input enters the equation in form of a summed current I_{syn}. To obtain its value, the time of each incoming spike is convolved with a kernel that represents the excursion of the membrane potential (*post-synaptic potential*). Frequently used functions for this kernel are α-shapes with varying time constants, exponentially decaying functions or delta pulses. Multiple inputs are lumped together into I_{syn} before propagation of the equation to the next time step. This approach is commonly referred to as *current-based* modeling.

Another way to model the influence of external input is the *conductance-based* approach. In contrast to integrating the inputs into a general current variable, the input instead influences the conductance of the membrane in this case. This is generally considered more realistic, but leads to a non-linear differential equation, as changes to the conductance

depend on the membrane potential and vice versa. Simulating these models is computationally more demanding than it is for the current-based approach.

Due to their property of integrating incoming spikes and firing one if a threshold is reached, the family of models described above is known under the name *integrate-and-fire neurons*.

2.1 Neuron dynamics

Substituting the resistor and the capacitor of the RC circuit shown in Figure 1 by the rise time τ_m (*membrane time constant*) and the capacitance C, we obtain the following equation for the membrane potential V of the standard integrate-and-fire neuron:

$$\frac{d}{dt}V = -\frac{V}{\tau_m} + \frac{1}{C}I \tag{1}$$

The input current I is the sum of the synaptic current and any external input. The α-shaped synaptic current as a function of time t for one incoming spike is given by:

$$\imath(t) = \hat{\imath}\frac{e}{\tau_\alpha} t e^{\frac{-t}{\tau_\alpha}}. \tag{2}$$

Here $\hat{\imath}$ is the peak value of the incoming spike and τ_α is the rise time. The inhomogeneous differential equation (2) (for simplicity we assume that $I == \imath$) is rephrased as a homogeneous system of differential equations (or matrix differential equation):

$$\frac{d}{dt}\begin{pmatrix} \frac{d}{dt}\imath + \frac{1}{\tau_\alpha}\imath \\ \imath \\ V \end{pmatrix} = \begin{pmatrix} -\frac{1}{\tau_\alpha} & 0 & 0 \\ 1 & -\frac{1}{\tau_\alpha} & 0 \\ 0 & \frac{1}{C} & -\frac{1}{\tau_m} \end{pmatrix} \cdot \begin{pmatrix} \frac{d}{dt}\imath + \frac{1}{\tau_\alpha}\imath \\ \imath \\ V \end{pmatrix} \tag{3}$$

For a fixed time step t it is now possible to solve the differential equation by calculation of the matrix exponential of the given matrix [RD99]. This way of propagating the model in time is particularly efficient because it consists only of a few multiplications.

Although this calculation is done only once for each linear neuron model in NEST during the implementation of the model, it is tedious and has to be done manually. With NESTML, all necessary factors for the time propagation for any given synaptic current and any linear differential equation can be calculated automatically, which solves one of the major obstacles for developing new neuron models in NEST.

3 Related Work

Various modeling languages for neurons and neural networks exist, each of which focusing on different aspects of neural modeling. Here, we describe the representative examples NineML and NeuroML in detail.

3.1 NineML

The Network Interchange for Neuroscience Modeling Language (NineML, [Ra11, Go11]) provides an unambiguous description of spiking neural networks for model sharing and re-use. NineML defines a common object model that describes the different elements of a model in a neuronal network. This object model corresponds to its abstract syntax, while XML is used as its concrete syntax.

NineML consists of two semantic layers: the abstract layer describes the core concepts of a model alongside its mathematical description, parameter and state variables and state update rules. The user layer allows the description of state or parameter variables and definition of initial or default values and units. Objects defined in the user layer can be re-used in different models, while model re-use in the abstract layer is not supported.

In the abstract layer each network element is represented by a `ComponentClass` composed of a `Dynamics`-block and a set of `Interfaces`. The `Dynamics` contain the internal model dynamics, e.g. state variables and update rules. The `Interfaces` contain the parameters that can be set from the user layer and ports for the communication with other network elements. The advantage of the `ComponentClass` is that it supports any kind of network element instead of just complete neuron or synapse models. The drawback is that the exact kind of model modeled by the `ComponentClass` is unknown. It could be a neuron, a synapse or an ion channel and the relation to domain concepts is hidden from the user.

To make NineML descriptions simulator agnostic, they only provide differential equations to describe the dynamics of a model. As the system itself chooses the solver for the dynamics, this might lead to the generation of unnecessarily complex and inefficient code for a specific simulator or to an inaccurate solution of the model equations. Expressing neuron dynamics as a finite-state automaton with regimes and transitions as in NineML works well to visualize them. However, for developing new neuron models and expressing complex relationships between states a procedural definition of the dynamics is more intuitive.

3.2 NeuroML

The model description language NeuroML [Gl10] is a description language for biophysically detailed neuron and neural network models and enables interoperability across multiple simulators. Neuron models in NeuroML can have complex morphologies, voltage- and ligand-gated conductances, and synaptic mechanisms. Network models contain the 3D positions of cells and synapses in the network.

NeuroML is optimized for complex compartmental models, but also supports simple point neurons like the leaky integrate-and-fire model (Section 2). However, more advanced types of point neuron models such as the exponential integrate-and-fire neuron [BG05] or the Izhikevich model [Iz03] are not fully supported yet. The language itself is split in three levels, each of which is responsible for describing a different scale of biological detail:

Level 1 describes the morphology of a neuron model using the sub-language MorphML. This contains the number and 3D position of compartments and their size and shape. Additionally, it provides mechanisms to store metadata.

Level 2 uses ChannelML to describe voltage-gated membrane conductances together with static and plastic synaptic conductance processes. It also extends level 1 descriptions by specifying the location and density of membrane conductances in the cell model.

Level 3 describes neural networks with 3D locations of individual neurons, synaptic connections between neurons (in projections) and external inputs via NetworkML.

NeuroML can define neuron models by using predefined elements for segments, channel mechanisms or synapse mechanisms. This results in compact and clear definitions of models by outsourcing and reusing mechanism definitions. On the other hand, the limited set of possible language elements reduces the expressiveness of NeuroML to models for which corresponding elements exist. Defining new mechanisms requires changes to the language definition itself.

3.3 XML as carrier language

Most of the established modeling languages use XML [Ye04] as their concrete representation, because an ecosystem of tools already exists and no additional lexers and parsers have to be developed to check syntactic correctness. However, this approach has two disadvantages: first, the verbosity of XML makes writing and reading models difficult for modelers [Ch01] and sophisticated tools are required for creating, visualizing and understanding more complex models. Second, the model descriptions have to be processed separately to ensure semantic correctness. An example for this is NineML's `MathInline` statement, which requires custom parsers to check the contained mathematical expressions for correctness. Listing 1 illustrates these two problems of XML using an excerpt of a NineML file.

```
1  ...
2  <Dynamics>
3    <StateVariable name="V" dimension="voltage" />
4    <StateVariable name="U" dimension="voltage␣per␣time" />
5    <Alias name="rv" dimension="none">
6      <MathInline>V*U</MathInline>
7    </Alias>
8    <Regime name="subthresholdRegime">
9      <TimeDerivative variable="U">
10       <MathInline>a*(b*V - U)</MathInline>
11     </TimeDerivative>
12     <TimeDerivative variable="V">
13       <MathInline>0.04*V*V + 5*V + 140.0 - U + iSyn</MathInline>
14     </TimeDerivative>
15   </Regime>
16 </Dynamics>
17 ...
```

Listing 1: Excerpt from a NineML file. To declare the simple mathematical expression $rv = V * U$, three lines of code are required (cf. lines 5-7). The `MathInline` element in line 13 contains only a string that cannot be checked for syntactic or semantic correctness with existing XML tools.

3.4 Simulators

Before the existence of general model description languages, simulators already had their own languages for specifying models. In the case of NEST (Section 1.2) this language so far is just plain C++ and the features provided by the simulator API. The remainder of this section introduces two other approaches for the definition of neuron and network models for completeness.

Brian [St14] is a simulator for spiking neural networks written entirely in Python. It uses code generation based on SymPy, NumPy and Cython to obtain reasonable performance, but lacks the facilities for running distributed simulations. Neuron models are defined by specifying the differential equations written in a text-based mathematical notation. However, as these definitions are ordinary Python strings, checking context conditions and semantically analyzing them is difficult. Unless own extensions to Brian are provided, it is up to the simulator to chose a solver method for the equations, which can have negative effects on accuracy or efficiency. Brian is mainly used for small-scale and exploratory simulations on laptops and workstations.

NEURON [HC97] is a simulator mainly for compartmental neuron models with biophysical properties. Neuron and synapse models can be defined with a set of graphical tools or using the custom programming language HOC. NEURON's focus is not on large-scale modeling, but on the simulation of very detailed neuron models on large computer clusters and supercomputers. In principle, it also supports simulations of large networks of simple neuron models, but falls behind the performance and memory footprint of simulators that are aimed specifically at these simulations.

4 Modeling spiking neurons with NESTML

NESTML consists of three modular and separately usable sub-languages, a symbol table and context conditions. These languages together form the NESTML domain specific language (DSL).

Procedural DSL (PL) defines the imperative logic of the model. PL also provides a library with methods for emitting messages, logging and working with buffer objects.

Units DSL (UL) enables defining and checking variables with physical units like Volt (V) and Ampere (A). UL also supports common magnitudes like mili (m) and pico (p).

Differential Equation DSL (DL) provides the possibility to define differential equations in the form of a string of math notation and analyze these equations.

NESTML separates model definition from simulator specific code and thereby allows the user to concentrate on the development of models instead of implementation details. Automatic analysis of differential equations simplifies the formulation of new models by outsourcing the task of finding an accurate solution to NESTML's infrastructure. This section introduces NESTML with the example of a simple integrate-and-fire neuron [RD99].

4.1 Basic design and definitions

The general syntax of NESTML is inspired by that of Python, which is widely known to researchers in the computational neuroscience community [Mu09, Da13]. This lowers the entry barrier for new users and improves comprehensibility of models. NESTML supports common data types like *integer*, *real* and *string* as well as physical data types with units provided by the PL. Variables are defined by stating the name followed by a type or unit.

```
① neuron iaf_neuron:

② state:
    y0, y1, y2, y3, V_m mV [V_m >= -99.0]
    # Membrane potential
    alias V_rel mV = V_m + E_L
  end

③ function set_V_rel(v mV):
    y3 = v - E_L
  end

④ parameter:
    # Capacity of the membrane.
    C_m       pF = 250 [C_m  > 0]
  end

⑤ internal:
    h   ms    = resolution()
    P11 real = exp(-h / tau_syn)
    ...
    P32 real = 1 / C_m * (P33 - P11)
                / (-1/tau_m - -1/tau_syn)
  end

⑥ input:
    spikeBuffer   <- inhibitory
                     excitatory spike
    currentBuffer <- current
  end

⑦ output: spike

⑧ dynamics timestep(t ms):
    if r == 0: # not refractory
      V_m = P30 * (y0 + I_e) + P31 *
            y1 + P32 * y2 + P33 * V_m
    else:
      r = r - 1
    end
    # alpha shape PSCs
    V_m = P21 * y1 + P22 * y2
    y1 = y1 * P11
    y0 = currentBuffer.getSum(t);
  end
end
```

Excerpt from the explicit ODE solution

Figure 2: Excerpt from the integrate-and-fire neuron expressed in NESTML. See https://github.com/nest/nestml for the complete neuron model description.

A neuron in NESTML is declared by the keyword `neuron` and a name (① in Figure 2). The name can be used to reference the model from other NESTML models. Each neuron is composed of blocks with definitions of *state* and *parameter* variables, *inputs* and *outputs*. A *dynamics* function is responsible for the behavior of the neuron when the model is simulated. All blocks in NESTML start with a colon and end with the keyword `end`

`state` ② contains the variables of the dynamic state of the neuron. An example for a state variable is the membrane potential of a neuron (`V_m`). An `alias` variable describes the dependency between variables using an expression (`V_rel`). For setting a value on an alias a setter function is required (`set_V_rel`), as the defining expression cannot be inverted automatically for the general case. Plausibility constraints can be added in square brackets after the variable definition (`V_m >= -99.0`). These are useful for debugging and during the development phase of the model and can be removed in the production version for better performance.

`parameter` ④ contains attributes that do not change over time, but may vary among neuron instances. Examples are the length of the refractory period or the membrane capacitance (`C_m`). To ensure that values are in a sensible range, it is possible to define guards which are evaluated every time a parameter is changed by the user. The syntax is the same as for the plausibility constraints in the `state` block.

`internal` ⑤ contains values that depend on the parameters, but can be precalculated once or auxiliary variables needed for the implementation. In Figure 2 for example, the propagator matrix (i.e. the solution of the model equation) is defined in this block.

`input` ⑥ Several named inputs can be declared using the name of the buffer that should receive the specified input during simulation. The input type can specified as `spike` or `current`. A spike input can further be `inhibitory`, `excitatory` or both. Depending on the sign of the input, incoming spikes are routed to the corresponding sub-buffer. If no such modifier is given the buffer receives all spikes.

`output` ⑦ Each neuron in NEST can just send one type of event during simulation. NESTML supports `spike` or `current` output, which is specified after the keyword `output`.

Functions allow the convenient reuse of code (e.g. ③ in Figure 2). Their definition starts with the keyword `function` followed by the function name and a list of zero or more function parameters in parentheses. Just like declaring a variable, a parameter is declared by first stating its name and then its type. Multiple parameters are separated by a comma. The parameter list is followed by an optional return type.

The definition of the dynamics of a neuron is similar to that of a function (e.g. ⑧ in Figure 2). It starts with the keyword `dynamics` followed by the type of the dynamics. Depending on the type, the function is called once per update step (`timestep`) or just once per minimum delay interval in the simulated network (`minDelay`). A list of parameters can be defined in parentheses.

4.2 Modularity and component concept

In order to reuse parts of a model they must be defined in a block starting with the keyword `component` and a name (② in Figure 3). The component is then `imported` into a neuron (see ①) and made available using the keyword `use` and optionally giving a convenient name (see ③). Functions and variables from the component can be referenced using the dot-notation (see ④).

```
① import PSPHelpers

  neuron iaf_neuron:

③   use PSPHelpers as PSP

    dynamics timestep(t ms):
④     PSP.computePSPStep(t)
      # alpha shape PSCs
      y2 = P21 * y1 + P22 * y2
      y1 = y1 * P11
    end

    ...

  end
```

```
② component PSPHelpers:
    state:
      - y0, y1, y2, V_m mV [V_m >= 0]
      alias V_rel mV = y3 + E_L
    end

    function computePSPStep(t ms):
      if r == 0: # not refractory
        y3 = P30 * (y0 + I_e) + P31 *
               y1 + P32 * y2 + P33 * y3
      else:
        r = r - 1
      end

    end
    ...
  end
```

Figure 3: An example for a neuron that reuses a function from a component. Left panel: the code of the referencing neuron; right panel: the code of the component.

This concludes the description of the imperative approach, where the solution of the underlying differential equation is described completely and explicitly in the blocks `internal` and `dynamics`. This approach maps directly to the current implementation of models in NEST. In addition, NESTML provides a declarative approach that is more intuitive, because it is closer to the mathematical description of neuron models common in computational neuroscience.

4.3 Declarative model definition

One of the main difficulties in writing models for NEST is writing the code for solving the equations, as this requires advanced knowledge of mathematics and numerics. In the declarative approach, differential equations are directly expressed as a string in mathematical notation under an `ODE` block. Figure 4 shows the declaration of an equation for the current (see ③) and the differential equation for the membrane potential `V_m` (see ④). As this is also the way how models are presented in publications, this syntax makes it easy to re-implement published models in NESTML.

```
neuron iaf_neuron:
  internal:
    h   ms   = resolution()
    P11 real = exp(-h / tau_syn)
    ...
    P32 real = 1 / C_m * (P33 - P11)
                 / (-1/tau_m - -1/tau_syn)
  end

  dynamics timestep(t ms):
    if r == 0: # not refractory
①     V_m = P30 * (y0 + I_e) + P31 *
              z1 + P32 * y2 + P33 * y3
    else:
      r = r - 1
    end
    # alpha shape PSCs
②   V_m = P21 * y1 + P22 * V_m
    y1 = y1 * P11
  end

end
```

```
neuron iaf_neuron_ode:
  internal:
    h   ms   = resolution()
  end

  dynamics timestep(t ms):
      if r == 0: # not refractory
        ODE:
③         I_shape == w * (E/tau_in) * t *
                     exp(-1/tau_in*t)
④         d/dt V_m == -1/Tau * V_m +
                      1/C_m*I_shape
        end
      else:
        r = r - 1
      end
    end

end
```

Figure 4: Modeling an integrate-and-fire neuron in NESTML. Left panel: using the imperative approach calculating `V_m` explicitly (see ① and ②). Right panel: just specifying the shape of the synaptic current (see ③) and the differntial equation for `V_m` (see ④).

Calculating the matrix for propagating the state is usually a time consuming manual task in NEST. The possibility to write models in a declarative fashion thus considerably reduces the work required to define new models. With the imperative approach still available, we don't have to sacrifice control over other parts of the neuron dynamics, which can nonetheless be expressed as procedural code.

The detailed mathematical and algorithmic techniques for transforming neural dynamics equations to efficient and accurate C++ code are out of scope of the current manuscript, and will be published in a follow-up article.

4.4 Implementation of NESTML

NESTML is implemented using the MontiCore [KRV08, KRV07] language workbench, which enables an agile development of DSLs. Based on a context-free grammar, MontiCore defines concrete and abstract representation as abstract syntax tree (AST) and provides infrastructure for checking the compliance to the rules via context conditions [Vö11]. MontiCore supports various mechanisms for heterogeneous language integration, e.g. language aggregation, inheritance, and embedding. These features were used for implementing the NESMTL language inheritance and embedding [Lo13].

The modular design of NESTML gives users the flexibility to exchanges parts of NESTML. For example it enables us to embed Python to be used instead of the Procedural DSL in the future.

MontiCore provides a symbol table infrastructure [Ha15]. The symbol table stores symbols of the model and provides them to the language mechanisms. An example are NESTML components, which provide available functions with their signature, but hide the implementation. NESTML's symbol table automatically handles the resolution of model elements distributed over several files.

All languages of NESTML are strongly typed to allow type compatibility checks within and between models. The checks are performed using information from the symbol table. A constraint inside a model could be one that checks if the `dynamics` block only changes values of the `state` block, while one between models could be a check if a function called from an imported component is actually defined there. The framework to check such context conditions is also provided by MontiCore.

5 NESTML Tool Support

We provide a command line interface to the NESTML tools. They process NESTML model descriptions by parsing them, checking context conditions on the described model and generating the C++ model implementation and bootstrapping code for NEST. The code generator is based on the MontiCore generation framework [Sc12], which uses exogenous model-based transformations [MVG06] to integrate the solution code for the differential equation and a template-based system [CH06]. After executing the NESTML tools on a model description, the generated code can be compiled and the model immediately be used in NEST.

During model processing (Figure 5) an abstract syntax tree (AST) is created from the source model and context conditions are checked. From the AST, a SymPy script [Sy14] is generated and executed later by the code generator. For linear neuron models, the script returns the matrix entries of the propagator matrix, or the right hand side of the ODE for use in a solver otherwise. The source AST is transformed by adding these entries as variable declarations to the `internal` block. The altered AST is serialized by pretty printing it again as a NESTML description. This way, the model developer can inspect

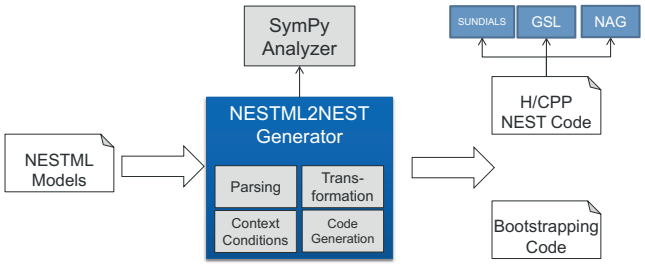


Figure 5: Processing of models in the NESTML frontend. Processing steps include parsing, checking context conditions, model transformations and code generation.

how the solution of the ODEs is implemented. If the model doesn't contain any differential equations, this step is skipped and code generation is performed on the initial AST.

The code generator produces C++ implementation and header files for each model. The integration code consists of a C++ file describing the module and a set of scripts for bootstrapping in NEST. depending on the SymPy analysis, the generated code either contains an explicit implementation of the solution for the ODE, or code that relays the right hand side of the ODE to a numerical solver, e.g. SUNDIALS [Hi05], GSL [Go09] or NAG [HF01].

The frontend is implemented as a regular Java archive available as a download from https://github.com/nest/nestml. It has the following modes of execution:

`parse` Models are parsed and syntactic correctness will be reported.
`contextConditions` Models are parsed and checked against the context conditions. Syntactic and semantic correctness will be reported.
`generate` The code generation workflow will be executed after parsing and checking context conditions.

6 Discussion and Outlook

We presented the NEST Modeling Language NESTML to describe spiking neurons and introduced a code generator for the NEST platform. NESTML supports two development paradigms: an imperative scheme based on a procedural language and a declarative scheme using a textual definition of differential equations. Both paradigms can be transparently combined in the same neuron model in order to increase the expressiveness. A dedicated module concept allows a seamless reuse of models and model components. The complexity of model development is decreased by abstracting the implementation and infrastructure details.

Using NESTML, neuron models can be described using domain concepts. The syntax of NESTML is similar to that of the Python programming language, which is well known to the computational neuroscience community. As NESTML is implemented as a MontiCore language, the development of language variants using language inheritance or language

embedding is straightforward [Lo13]. This fact can be exploited in the future to develop a family of NESTML languages targeting users with different technical expertise or adding code generation for other simulators.

In order to demonstrate the usefulness of the proposed language, we reformulated about 30% of NEST's models as NESTML in a collaboration with the developers of NEST. The language and tools were generally well received and especially the concise syntax and the code generation pipeline were mentioned as big improvements. NESTML provides a 20 fold reduction of code between model description and generated implementation code. This value includes both the C++ model code as well as the bootstrapping code.

Large-scale modeling of nervous systems requires an abstraction as provided by NESTML to increase modeling capabilities, reusability and maintainability. This is an interesting challenge from the viewpoint of software language engineering and ongoing research will show, how to raise the level of modeling capabilities even further.

NESTML is publically available on https://github.com/nest/nestml. Our ongoing work focuses on the addition of features for the description of synapse models in NESTML. The language and the tools will be evaluated in a more structured way at a community workshop this winter and the feedback will be incorporated into the next public release of NESTML.

Acknowledgments This work is supported by the JARA-HPC Seed Fund *NESTML - A modeling language for spiking neuron and synapse models for NEST*, the *Initiative and Networking Fund* of the Helmholtz Association and the Hemholtz Portfolio Theme *Simulation and Modeling for the Human Brain*. We gratefully acknowledge fruitful discussions with Markus Diesmann and Hans Ekkehard Plesser.

References

[BG05] Brette, Romain; Gerstner, Wulfram: Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity. J Neurophysiol, 94(5):3637–3642, 2005.

[Ch01] Cheney, J.: Compressing XML with multiplexed hierarchical PPM models. In: Data Compression Conference, 2001. Proceedings. DCC 2001. pp. 163–172, 2001.

[CH06] Czarnecki, K.; Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Syst. J., 45(3):621–645, July 2006.

[Cr12] Crook, Sharon; Bednar, James; Berger, Sandra; Cannon, Robert; Davison, Andrew; Djurfeldt, Mikael; Eppler, Jochen; Kriener, Birgit; Furber, Steven; Graham, Bruce; Plesser, Hans Ekkehard; Schwabe, Lars; Smith, Leslie; Steuber, Volker; van Albada, Sacha: Creating, Documenting and Sharing Network Models. Network: Computation in Neural Systems, 23:131–149, 2012.

[Da13] Davison, Andrew P.; Diesmann, Markus; Gewaltig, Marc-Oliver; Ghosh, Satrajit S.; Perez, Fernando; Muller, Eilif Benjamin; Bednar, James A.; Thirion, Bertrand; Halchenko, Yaroslav O.: Research topic: Python in Neuroscience II. Frontiers in Neuroinformatics, 2013. http://journal.frontiersin.org/researchtopic/1591.

[GD07] Gewaltig, Marc-Oliver; Diesmann, Markus: NEST (NEural Simulation Tool). Scholarpedia, 2(4), 2007.

[Gl10] Gleeson, Padraig; Crook, Sharon; Cannon, Robert C.; Hines, Michael L.; Billings, Guy O.; Farinella, Matteo; Morse, Thomas M.; Davison, Andrew P.; Ray, Subhasis; Bhalla, Upinder S.; Barnes, Simon R.; Dimitrova, Yoana D.; Silver, R. Angus: NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail. PLoS Comput Biol, 6(6), 06 2010.

[Go09] Gough, Brian: GNU scientific library reference manual. Technical report, 2009.

[Go11] Gorchetchnikov, Anatoli; Raikov, Ivan; Hull, Mike; Le Franc, Yann: Network Interchange for Neuroscience Modeling Language (NineML) – Specification. Technical report, INCF Task Force on Multi-Scale Modeling, July 2011. http://software.incf.org/software/nineml/wiki/nineml-specification/.

[Ha15] Haber, Arne; Look, Markus; Mir Seyed Nazari, Pedram; Navarro Perez, Antonio; Rumpe, Bernhard; Völkel, Steven; Wortmann, Andreas: Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In (Hammoudi, Slimane; Pires, Luis Ferreira; Desfray, Philippe; Filipe, Joaquim Filipe, eds): Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development. SciTePress, Angers, Loire Valley, France, pp. 19–31, February 2015.

[HC97] Hines, M. L.; Carnevale, N. T.: The NEURON Simulation Environment. Neural Computation, 9(6):1179–1209, aug 1997.

[He12] Helias, Moritz; Kunkel, Susanne; Masumoto, Gen; Igarashi, Jun; Eppler, Jochen Martin; Ishii, Shin; Fukai, Tomoki; Morrison, Abigail; Diesmann, Markus: Supercomputers ready for use as discovery machines for neuroscience. Front. Neuroinform., 6:26, 2012.

[HF01] Hoffman, Joe D; Frankel, Steven: Numerical methods for engineers and scientists. CRC press, 2001.

[HH52] Hodgkin, A. L.; Huxley, A. F.: A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve. 117:500–544, 1952.

[Hi05] Hindmarsh, Alan C; Brown, Peter N; Grant, Keith E; Lee, Steven L; Serban, Radu; Shumaker, Dan E; Woodward, Carol S: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. ACM Transactions on Mathematical Software (TOMS), 31(3):363–396, 2005.

[Iz03] Izhikevich, Eugene M et al.: Simple model of spiking neurons. IEEE Transactions on neural networks, 14(6):1569–1572, 2003.

[KRV07] Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. volume 4735 of LNCS, Nashville, TN, USA, pp. 286–300, October 2007.

[KRV08] Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: Monticore: Modular Development of Textual Domain Specific Languages. volume 11 of LNBIP, Zurich, Switzerland, pp. 297–315, July 2008.

[Ku14] Kunkel, Susanne; Schmidt, Maximilian; Eppler, Jochen Martin; Plesser, Hans Ekkehard; Masumoto, Gen; Igarashi, Jun; Ishii, Shin; Fukai, Tomoki; Morrison, Abigail; Diesmann, Markus; Helias, Moritz: Spiking network simulation code for petascale computers. Frontiers in Neuroinformatics, 8(78), 2014.

[La07] Lapicque, L.: Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. J. Physiol. Pathol. Gen, 9:620–635, 1907.

[Lo13] Look, Markus; Navarro Pérez, Antonio; Ringert, Jan Oliver; Rumpe, Bernhard; Wortmann, Andreas: Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In (Combemale, B.; De Antoni, J.; France, R. B., eds): Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC). volume 1102 of CEUR Workshop Proceedings, Miami, Florida, USA, 2013.

[MP43] McCulloch, Warren S.; Pitts, Walter: A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

[Mu09] Muller, Eilif; Bednar, James A.; Diesmann, Markus; Gewaltig, Marc-Oliver; Hines, Michael; Davison, Andrew P.: Research topic: Python in Neuroscience. Frontiers in Neuroinformatics, 2009. http://journal.frontiersin.org/researchtopic/8.

[MVG06] Mens, Tom; Van Gorp, Pieter: A Taxonomy of Model Transformation. Electron. Notes Theor. Comput. Sci., 152:125–142, March 2006.

[Ni01] Nicholls, John G.; Martin, Robert A.; Wallace, Bruce G.; Fuchs, Paul A.: From neuron to brain. Sinauer Associates, Sunderland, MA, 4 edition, 2001.

[Ra11] Raikov, Ivan; Cannon, Robert; Clewley, Robert; Cornelis, Hugo; Davison, Andrew; De Schutter, Erik; Djurfeldt, Mikael; Gleeson, Padraig; Gorchetchnikov, Anatoli; Plesser, Hans Ekkehard; Hill, Sean; Hines, Mike; Kriener, Birgit; Le Franc, Yann; Lo, Chung-Chan; Morrison, Abigail; Muller, Eilif; Ray, Subhasis; Schwabe, Lars; Szatmary, Botond: NineML: the network interchange for neuroscience modeling language. BMC Neuroscience, 12(Suppl 1):P330, 2011.

[RD99] Rotter, Stefan; Diesmann, Markus: Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. Biological cybernetics, 81(5-6):381–402, 1999.

[Sc12] Schindler, Martin: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sh91] Shepherd, Gordon M.: Foundations of the Neuron Doctrine. Oxford Univ. Press, 1991.

[St14] Stimberg, Marcel; Goodman, Dan F. M.; Benichoux, Victor; Brette, Romain: Equation-oriented specification of neural models for simulations. Frontiers in Neuroinformatics, 8(6), 2014.

[Sy14] SymPy Development Team: . SymPy: Python library for symbolic mathematics, 2014. http://www.sympy.org.

[Vö11] Völkel, Steven: Kompositionale Entwicklung domänenspezifischer Sprachen. Aachener Informatik-Berichte, Software Engineering 9. Shaker Verlag, 2011.

[Ye04] Yergeau, François; Bray, Tim; Paoli, Jean; Sperberg-McQueen, C Michael; Maler, Eve: Extensible markup language (XML) 1.0. W3C Recommendation, 4, 2004.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 109

Infrastructure to Use OCL for Runtime Structural Compatibility Checks of Simulink Models

Vincent Bertram[1] , Peter Manhart[2] , Dimitri Plotnikov[1], Bernhard Rumpe[1], Christoph Schulze[1] and Michael von Wenckstern[1]

Abstract: Functional development of embedded software systems in the automotive industry is mostly done using models consisting of highly adjustable and potentially reusable components. A basic pre-requisite for reuse is structural compatibility of available component versions and variants. Since each vendor in the automotive domain uses its own toolchain with corresponding models, an unified modeling notation is needed. For this reason based on a detailed feature analysis of well-established and commonly used modeling languages, a meta-model has been derived that allows checking structural compatibility, even between heterogeneous modeling languages.

Keywords: Automotive, C&C ADLs, OCL, Meta-Model, Simulink, Structural Compatibility

1 Introduction

Nowadays, the differentiation of vehicles will take place not just in body and interior design but also in embedded software systems like adaptive cruise control or road sign detection. The growing number of Advanced Driver Assisted Systems (ADAS) and their emerging variants and versions each using individual components cause an increase in software maintenance costs making up to 60% of total software costs [Gl01]. Reusing software components reduces development and maintenance costs. A basic precondition is component compatibility. Structural compatibility serves as a first indicator as it is an important prerequisite for full compatibility, which would also enclose behavioral compatibility [Ru15], also called refinement [Ru96].

First, the methodology and infrastructure to check structural compatibility will be introduced. In Sec. 3 a meta-model is proposed that is based on a detailed feature analysis of well-established and commonly used modeling languages and their constraints. In Sec. 4 the feasibility of the proposed overall approach is demonstrated using an ADAS model developed in the industrial research project SPES_XT. This paper finishes with related work in Sec. 5 as well as a conclusion and outlook in Sec. 6.

2 Method to Check Structural Compatibility

To define structural compatibility constraints at runtime, the first order predicate logic language OCL [Ob14] is chosen to describe meta-model constraints. One of its benefits

[1] RWTH Aachen University, Chair of Software Engineering, Ahornstraße 55, 52074 Aachen
[2] Daimler Research & Development Ulm, RD/EEC, Wilhelm-Runge-Straße 11, 89081 Ulm

is that it is reasonable efficiently executable. The *Java*-based derivate OCL/Programmable (OCL/P) is used; as it is based on an easier syntax and thus easier to read and understand [Ru11, Ru12] compared to OMG's OCL.

In order to express properties of widespread Component & Connector (C&C) architectures, a generic meta-model which is encoded as a Class Diagram (CD) was developed. CDs encapsulate the state and functions of objects in form of attributes and methods. They are well suited to define the meta-model describing the data structure of C&C architectures to check compatibility on. The semantics of a CD [HR04, CGR08] is a set of valid objects. Since the meta-model describes the signature of C&C models in a notation independent manner, every C&C model can be transformed into an Object Diagram (OD) instance being conform to the meta-model [MRR11]. OCL/P constraints are then used to define structural compatibility between two of these ODs.

The overall idea of the proposed infrastructure is to check compatibility using a Satisfiability Modulo Theories (SMT) solver. Thus, the meta-model, the used C&C model, and corresponding OCL/P constraints must be mapped to solver code. In a first step two *Simulink* [DH14] model components (SLC)s are transformed to ODs using the *MATLAB-Connector* API. Next the resulting ODs, the meta-model and the defined compatibility constraints are mapped to SMT code. Furthermore, the compatibility constraints are defined and stored in separated artifacts decoupling them from the C&C models. After merging all parts to one file artifact using a PartMerger [Gr15], Microsoft's Z3 solver [MB08] is invoked with the resulted SMT file. The solver can deal with uninterpreted functions to generate counterexample witnesses to show incompatibilities.

3 Meta-Model for Component and Connector Languages

In order to express OCL constraints for heterogeneous C&C architectures in a uniform way, this section defines an expressive meta-model. Existing meta-models were not capable to express all necessary aspects that were needed to check structural compatibility. Its syntax is derived from the results of an intensive analysis of the most important Architecture Description Languages (ADLs) used in the automotive domain. This section's outline is top-down: First, the most specific meta-model classes such as `FunctionComponentElement` and `FunctionComponent` are presented. Then, general classes like `TypeReference` and `Port` are described. Finally `DataType` and `Unit` are introduced.

FunctionComponentElement: The interface `FunctionComponentElement` is realized as composite design pattern to support hierarchical C&C models wherein the class `FunctionComponent` contains components and the class `PortConnector` associates two `Port` classes allowing directed communication from `source` to `dest` `Port` (c.f. Fig. 1a). The class `Interface` shown in Fig. 1b uses in- and out-ports for *direct* asynchronous communication. *Indirect* communication takes place over shared `GlobalStorage` elements.

TypeReference: The `DataType` and the `TypeReference` meta-model are adoptions from *EAST-ADL* [EA13] coupled with *SysML*. Properties being necessary for a concrete usage of a data type are split up for easier reuse. In order to be compatible with *Simulink*,

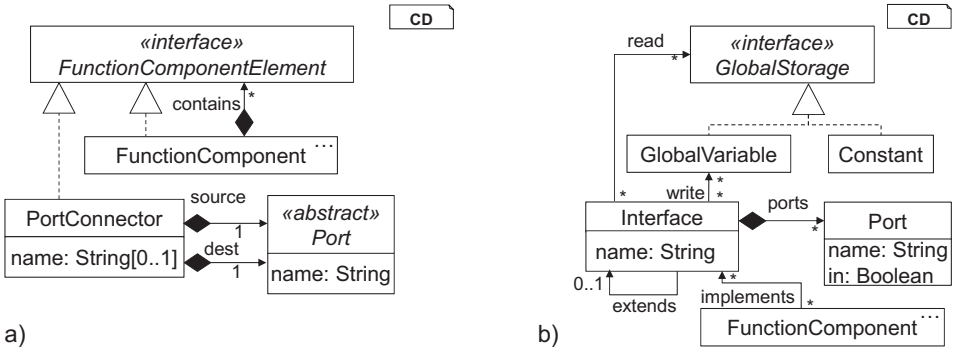


Fig. 1: Meta-Model of `FunctionComponentElement` and `FunctionComponent`

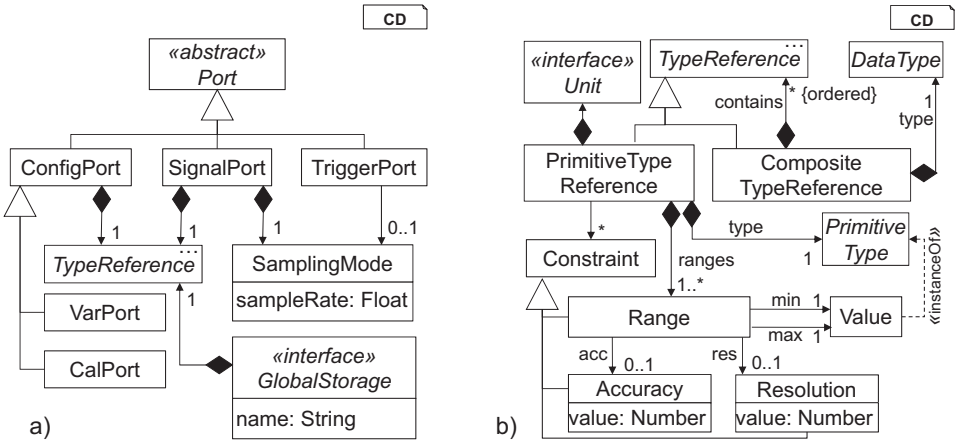


Fig. 2: Meta-Model of `Port`, `GlobalStorage` and `TypeReference`

changes were made on *EAST-ADL*'s homogeneous container having now a fixed amount of elements. A *Simulink* component that needs access to global variables is intentionally not supported by *AUTOSAR* [AU08], *EAST-ADL* and *MontiArc* [HRR12]. As a result *SysML*'s component interface model is used.

Port: The meta-model distinguishes three different kinds of `Port` classes (see Fig. 2a) based on the signal's purpose. This is similar to *AUTOSAR*. The class `TypeReference` of a `Port` defines the kind of signal content. Each `TypeReference` has at least one `Range` specifying the operative minimum and maximum (see Fig. 2b). If a signal has one range with higher accuracy (e.g. at low speed) than in another range (e.g. at high speed), there exists the possibility to define as many `Ranges` with its own `Accuracies` and `Resolutions` as necessary. The difference between `Resolution` and `Accuracy` is that `Resolution` represents the maximum delta how measured data can be stored and `Accuracy` represents the maximum error on stored data. Each `SignalPort` has additionally a `SamplingMode` describing the sample rate of a physical signal sampled into a digital one. If `TriggerPort` has a `SamplingMode`, an event can only arise at a specific time interval.

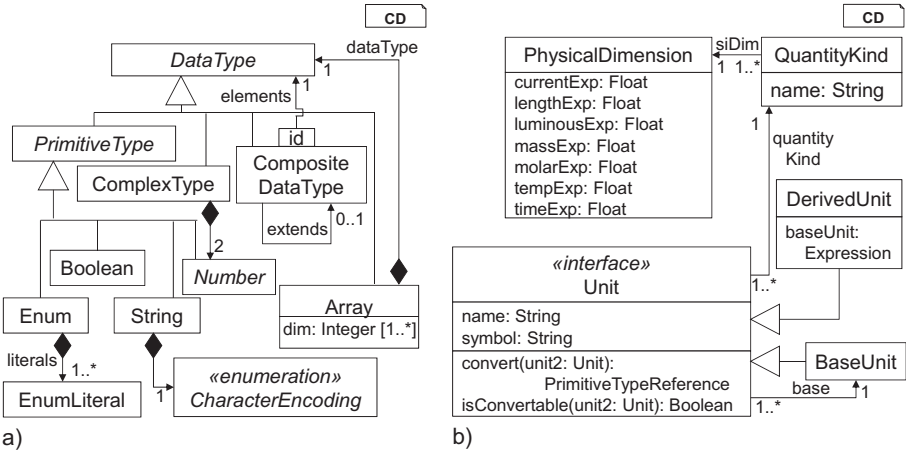


Fig. 3: Meta-Model of `DataType` and `Unit`

DataType: The class `DataType` shown in Fig. 3a is based on the type systems used by *Java* and *Simulink* where `ComplexType` represents complex numbers like $5+3i$, a `CompositeType` is a heterogeneous container accessing its children by an unique `id` and an `Array` is a homogeneous container of a specified length. An `Enum` consists of finite set of `EnumLiteral` elements. Each `String` has a specific `CharacterEncoding`. Nested ports that are available in *SysML* are similar to ports having a `CompositeType` as `DataType`.

Unit: The interface `Unit` in Fig. 3b is based on the unit type system of *SysML* [Ha06] and *Modelica* [Mo12]. `Unit` has a `QuantiyKind` such as acceleration, energy or speed. Each `QuantiyKind` has a physical dimension defined in terms of basic units. Two different `QuantityKind` objects can have the same `PhysicalDimension`, e.g. *torque* (*Nm*) and *energy* (*J*) are different units, but have the same physical dimension $\frac{kg \cdot m^2}{s^2}$. The class `DerivedUnit` specifies how units with different magnitudes within the same `QuantiyKind` are converted by stating a conversion formula. Among other earlier mentioned ADLs, the proposed meta-model allows to create all existent units, supporting either the metrical (km/h) or empirical (mph) measurement system.

4 ADAS Component Demonstrating Methodology

This section starts with an overview of the evaluated *ADAS_V1* component and continues with the translation of structural information of SLCs into the proposed meta-model.

The provided ADAS model has four different levels of evolution and is realized as *Simulink* model which is a special case of an C&C architecture software mainly but not only used in the automotive domain. This section depicts the structure of the *ADAS_V1*'s top-level SLC only. The running example is simplified and does not represent a 100% real world model, but it provides enough structural information without being overloaded with unimportant

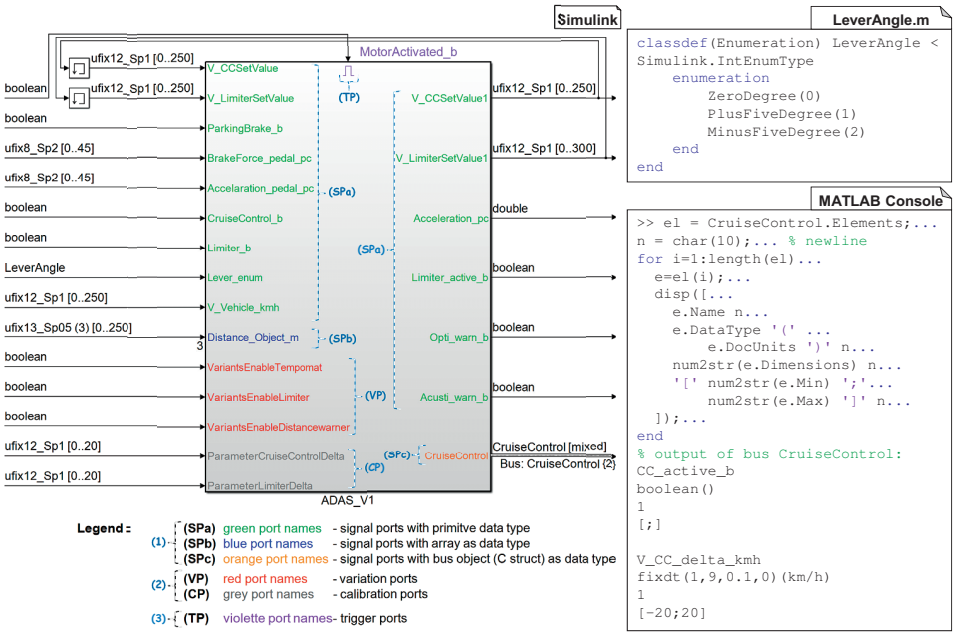


Fig. 4: Syntactic Interface Description of *ADAS_V1* developed in the project SPES_XT

details. The whole SLC interface is defined as the set of all in-, out-ports as well as used variables being shared outside this component.

The SLC interface of the *ADAS_V1* as shown in Fig. 4 is clearly evident, because it is completely covered by all visible in- and out-ports. This example consists of three different port types: (1) signal, (2) configuration, and (3) trigger ports. Signal ports can be grouped using arrays or non-virtual buses and have been divided into three subtypes: (SPa) for one single primitive type, (SPb) for grouped signals having the same data type, and (SPc) for grouped signals of different data types. Due to different purposes for using a configuration port, it is divided into two subtypes: (VP) for software variation (enabling or disabling features) and (CP) for calibrating subcomponents with parameters. The trigger port (TP) has only one kind. All port types (CP, SP, TP, VP) extend the port concept defined in *AUTOSAR*. The ports' data types with its ranges together are displayed directly on the signals connected to them; `ufix12_Sp1 [0..250]` stands for unsigned fixed point data type of word length 12 and having a slope (resolution) of 0.1 and a value range between 0 and 250.

In the example SLC of *ADAS_V1*, the port `Lever_enum` is an enumeration of `LeverAngle` and contains three values representing the deflecting angle of the lever: `ZeroDegree`, `PlusFiveDegree`, and `MinusFiveDegree` (c.f `LeverAngle.m` in Fig. 4). Non-virtual data types represent a semantic union. The non-virtual data type `CruiseControl` groups the boolean signal `CC_active_b` and the fix-point number signal `V_CC_delta_kmh` having a $[-20;20]$ range together to a non-virtual bus (c.f. `MATLAB Console` in Fig. 4).

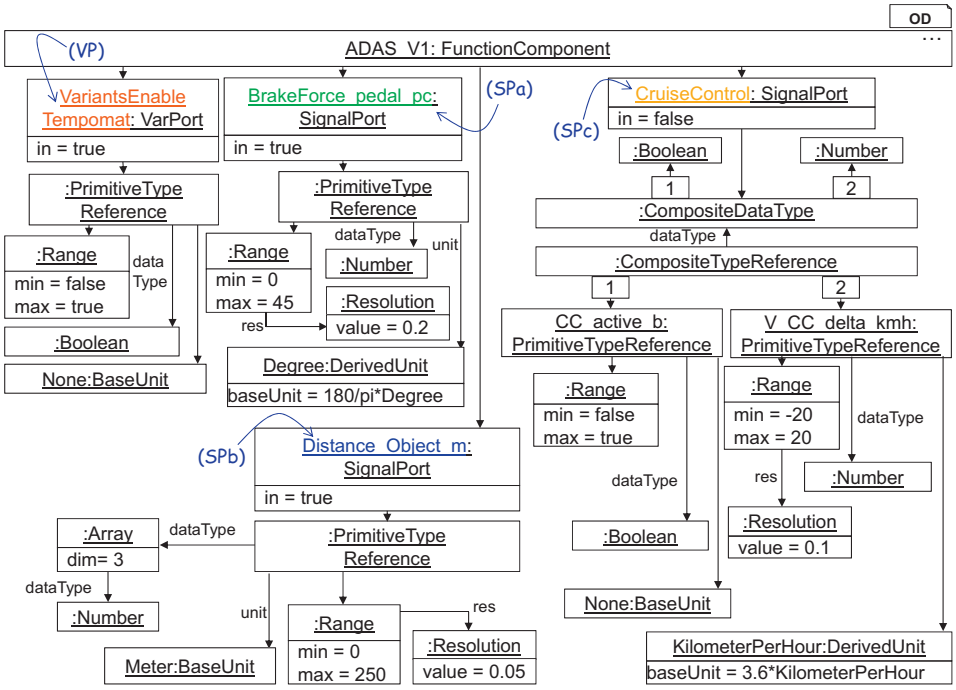


Fig. 5: Example meta-model instantiation of *ADAS_V1* Limiter component

```
OCL/P
1 def Boolean infix (Number v) isin (Range r) is:
2   result =
4     v >= r.min &&
5     v <= r.max &&
6     (~r.res ||
7       (v - range.min) % range.res == 0)
```

(Range r has no optional association res to Resolution)

Fig. 6: OCL code snippet and SMT function defining whether `Number` v belongs to a given `Range` r

To show an example instantiation of the meta-model, a subset of *ADAS_V1* is chosen and shown as an OD in Fig. 5. The excerpt uses 3 of 15 in-ports (`VariantsEnableTempomat` (VP), `BrakeForce_pedal_pc` (SPa), `Distance_Object_m` (SPb)) and 1 of 7 out-ports (`CruiseControl` (SPc)) containing all port types already shown in Fig. 2. The classes `CompositeDataType` and `CompositeTypeRefence` use numbers instead of names as identifiers as this allow an easier iteration over the elements in the used OCL constraints.

The code shown in Fig. 6 defines the semantics for the overloaded `isin` operator in OCL. This operator returns the `Boolean` value `true` when the `Number` `v` can be found in the given `Range` `r`, which is specified by a minimum (`min`), maximum (`max`) and a resolution (`res`). As an example a range with the properties `min=1.0, max=4.1, res=0.5` would return `true` only for the values `1.0, 1.5, 2.0, 2.5, 3.0, 3.5` and `4.0`, which is

defined in the lines 4, 5 and 7. Line 6 is only necessary because the `res` association is optional. If it is not defined, the range will contain all values between `min` and `max`.

5 Related Work

MontiArc [HRR12] is a C&C ADL where asynchronous message based communication is done over directed connectors between typed component ports. It uses context conditions for structural consistency checks such as data types and input/output directions of ports. Context conditions extend the expressiveness of grammar based approaches and enable checking of constraints, e.g. type systems, which are not expressible through grammars.

The approach presented in [Bh11] tackles the problem of defining and evaluating consistency relation between architectural views imposed by various heterogeneous models and a base architecture for the concrete system model. The consistency check happens on different abstraction levels if the concrete architecture model is consistent to the more abstract and less detailed abstract view. Instead, this approach checks compatibility on the same abstraction level.

The approach from [Da14] presents an architectural framework where multiple views on one system can be defined. The consistency is checked by leaving out details unnecessary for comparison, which is termed as a "lifting" operation. It lifts the more detailed software view to the more abstract functional view. This lifting operation causes valuable information loss, so the architectural framework is insufficient to check the structural compatibility as described in Sec. 2. The *SysML* meta-model which is used, supports only a subset of features available in the meta-model presented in Sec. 3.

6 Conclusion and Outlook

This paper has presented a first overview of an infrastructure for compatibility constraint checks using OCL. It was exemplified using a model from automotive domain but it is not limited to only this. Comparatively *MATLAB Simulink* is also used in other domains like aerospace and medical engineering. The main contribution is the proposed meta-model which is based on an intensive analysis of well-established modeling languages to support heterogeneous C&C architectures. It includes all meta-elements of the most known industrial meta-models (SysML, Simulink, Modelica, AUTOSAR, EAST-ADL). All of these ADLs can be mapped and as a result only a meta-model transformation must be written.

While this paper deals with the modeling aspect of the proposed infrastructure, future work will give more information about the generative part. This will contain how to generate SMT code for the Z3 solver using OCL/P constraints. The solver results will be presented as user-friendly error messages describing the reason for constraint violations based on counter-example witnesses. The highly modular infrastructure is capable of supporting new third party plug-ins and consequently allows a seamless integration into industrial development environments. To demonstrate the extensibility of the proposed infrastructure further modeling notations will be implemented.

References

[AU08] AUTOSAR: SW-C and System Modeling Guide. Technical report, 2008.

[Bh11] Bhave, A.; Krogh, B.H.; Garlan, D.; Schmerl, B.: View Consistency in Architectures for Cyber-Physical Systems. In: ICCPS. IEEE, 2011.

[CGR08] Cengarle, María Victoria; Grönniger, Hans; Rumpe, Bernhard: System Model Semantics of Class Diagrams. Technical report, TU Braunschweig, 2008.

[Da14] Dajsuren, Yanja; Gerpheide, Christine M.; Serebrenik, Alexander; Wijs, Anton; Vasilescu, Bogdan; van den Brand, Mark G.J.: Formalizing Correspondence Rules for Automotive Architecture Views. In: QoSA. ACM New York, 2014.

[DH14] Dabney, James B.; Harman, Thomas L.: Mastering Simulink. Prentice Hall, 2014.

[EA13] EAST-ADL: EAST-ADL Domain Model Specification. Technical report, 2013.

[Gl01] Glass, Robert L.: Frequently Forgotten Fundamental Facts About Software Engineering. IEEE Software, 2001.

[Gr15] Greifenberg, Timo; Hölldobler, Katrin; Kolassa, Carsten; Look, Markus; Mir Seyed Nazari, Pedram; Müller, Klaus; Navarro Perez, Antonio; Plotnikov, Dimitri; Reiss, Dirk; Roth, Alexander; Rumpe, Bernhard; Schindler, Martin; Wortmann, Andreas: A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development. 2015.

[Ha06] Hause, Matthew: The SysML modelling language. In: SysCon. 2006.

[HR04] Harel, David; Rumpe, Bernhard: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer, 2004.

[HRR12] Haber, Arne; Ringert, Jan Oliver; Rumpe, Bernard: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical report, RWTH Aachen, 2012.

[MB08] Moura, Leonardo; Bjørner, Nikolaj: Z3: An Efficient SMT Solver. In: TACAS. volume 4963 of LNCS. Springer, 2008.

[Mo12] Modelica Association: Modelica - A Unified Object-Oriented Language for Systems Modeling. Technical report, 2012.

[MRR11] Maoz, Shahar; Ringert, Jan Oliver; Rumpe, Bernhard: Modal Object Diagrams. In: ECOOP. volume 6813 of LNCS. Springer, 2011.

[Ob14] Object Management Group: Object Constraint Language, Version 2.4. Technical report, 2014.

[Ru96] Rumpe, Bernhard: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Herbert Utz Verlag, 1996.

[Ru11] Rumpe, Bernhard: Modellierung mit UML. Springer, 2011.

[Ru12] Rumpe, Bernhard: Agile Modellierung mit UML. Springer, 2012.

[Ru15] Rumpe, Bernhard; Schulze, Christoph; Wenckstern, Michael von; Ringert, Jan Oliver; Manhart, Peter: Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In: SPLC. ACM New York, 2015.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 117

Towards a Catalog of Structural and Behavioral Verification Tasks for UML/OCL Models

Frank Hilken[1], Philipp Niemann[1], Martin Gogolla[1], Robert Wille[2]

Abstract: Verification tasks for UML and OCL models can be classified into structural and behavioral tasks. For both task categories a variety of partly automatic solving approaches exist. But up to now, different interpretations of central notions as, for example, 'consistency' or 'reachability' can be found in current approaches and tools. This paper is designed to clarify central verification notions and to establish a collection of typical verification tasks that are common to multiple approaches and tools. In addition, the verification tasks are categorized with the aim of creating a central catalog of tasks, providing a common understanding of the terms used in model verifications.

1 Introduction

The increasing usage of modelling languages like the *Unified Modelling Language* (UML) and the *Systems Modeling Language* (SysML) and their formalizations have lead to a variety of verification engines for various model descriptions. Along with these tools, a heap of verification tasks were created and defined, each approach with their own definitions. This process has lead to model verification terms, such as, *consistency* or *reachability*, that are used multiple times with differing semantics [CCR08].

In order to establish a general terminology and create a common understanding, this paper takes frequently used verification tasks, describes their goals and categorizes them into a catalog. The categories give a general idea and quick overview of the goals of the tasks assigned to them. In addition, these categories and tasks are divided into structural and behavior topics. The descriptions shall clarify the interpretation of verification tasks. Similar, distinct tasks were given concrete names and descriptions to seperate their overlap. For example, the *consistency* task was split into a *weak* and a *strong* consistency.

The goal of the catalog is a common understanding of the various existing verification tasks to reduce misinterpretations and establish a foundation for communicating about them with a clear understanding of their semantics. The catalog provided in this paper is not meant as a final product, but rather a basis to discuss and extend it.

The remainder of this paper is structured as follows: Section 2 pictures the state of the art and further motivates the categorization of verification tasks. Section 3 introduces a short running example that is used to exemplify the goal of selected verification tasks. Section 4

[1] University of Bremen, Computer Science Department, D-28359 Bremen, Germany
Email: {fhilken|pniemann|gogolla}@informatik.uni-bremen.de
[2] Johannes Kepler University, Computer Science Department, A-4040 Linz, Austria
Email: robert.wille@jku.at

first defines a metamodel to represent verification tasks and then finished with the actual catalog, categorizing and describing the verification tasks. Section 5 wraps up the paper with a conclusion.

2 Motivation

Modeling languages such as the *Unified Modeling Language* (UML) or the *Systems Modeling Language* (SysML) together with textual constraints, e.g., provided by the *Object Constraint Language*, have been established to specify the design of complex systems. They provide different concepts such as class diagrams, sequence diagrams, or activity diagrams which are expressive enough to formally specify a complex system, but hide specific implementation details. Since modeling languages permit formal descriptions, they additionally enable the verification of the respective specification already in the absence of a specific implementation[1].

The corresponding verification tasks can be divided into

- *Structural Verification Tasks*, where a single system state is considered, as well as
- *Behavioral Verification Tasks*, where a sequence of system states as well as their transitions (e.g., described by operations with pre- and postconditions) is considered.

For both categories of verification tasks, a variety of (automatic) solving approaches have been introduced in the recent past [CCR08, SWD11, An07, Ba12, CKZ11, EW04, La07, Ro14]. However, until today different interpretations and terminologies exist for the respectively considered verification tasks.

For structural verification tasks, definitions as proposed e.g. in [GKH09] became rather established. Nonetheless, even in this context, multiple notions and variations can be found in the literature. For instance, consider the well-established task of checking "consistency", i.e. investigating whether a model description is consistent in that sense that an instantiation of the model exists which satisfies all of the model's constraints: in [CCR08], any non-empty instantiation of the model is accepted, while [GHH14] forces each class of the model to be instantiated at least once.

For behavioral verification tasks, so far a comprehensive list of tasks has not even been attempted at all (to the best of our knowledge). In contrast, for other areas of validation and verification in modeling, similar compilations of verification tasks and techniques have been presented, e.g. a survey on tasks for model transformations in [CS13] or a survey on modeling techniques for behavioral verification of software product lines in [Be15].

In this work, we aim for providing a unique and clear definition of important verification tasks that can be applied on UML/OCL models. This includes a comprehensive consideration of both, structural and behavioral issues. Thus far, verification tasks are often

[1] In model-driven engineering, it is common to apply so called model transformations to automatically transform models into a different description mean or language during the design process. In this context, it is an important task to validate/verify whether source and target model of a transformation are equivalent. However, this is out of the scope of the present paper where we focus on the verification of stand-alone model descriptions

referenced using different terms or using the same term, but having in mind different meanings. By presenting a fine granular differentiation of tasks in the following sections, we try to reduce misinterpretations and establish a common basis for an improved and clarified communication about verification tasks.

3 Traffic Light Running Example

In this section we introduce a simplified pedestrian traffic light preemption which will serve as a running example to illustrate concepts discussed in the next section. The corresponding model is depicted in Fig. 1. The main class of the example is the `Controller` which is connected to exactly two traffic light signals, one for cars (`carLight`) and one for pedestrians (`pedLight`). For simplicity, we assume two-state signals (green light on/off). With the operation `pedRequest()`, pedestrians express the desire to cross the road from either side. The controller stores these requests in the `request` attribute and switches the corresponding signals using the `switchPedLight()` and `switchCarLight()` operations. To prevent accidents, the invariant `safety` ensures that pedestrians and cars may not both face a green light (indicating a safe crossing) at the same time.

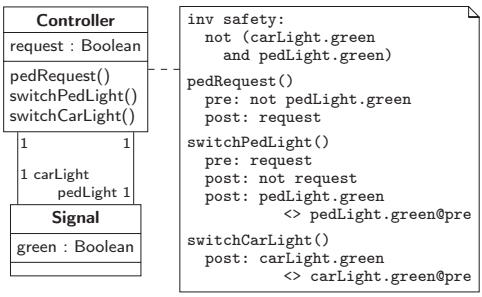


Fig. 1: Traffic light running example.

4 Categorizing Verification Tasks

We have identified a variety of verification tasks that are used in model checkers and extracted use cases from them. These use cases were assigned to five basic categories, giving a quick overview of the general goal of each task. The five categories are *Consistency*, *Independence*, *Reachability*, *Executability* and *Consequence*. The *Consistency* category represents general instantiability use cases, the *Independence* category describes use cases checking relations of model elements, the *Reachability* category contains use cases that check if certain goals are reachable when the behavior of the model is simulated, the *Executability* category specifies use cases that examine possible transitions between system states and, finally, the *Consequence* category characterizes use cases that deduct model properties and put model elements into relation. These categories naturally divide into two areas: *structural* and *behavioral* tasks.

Figure 2 illustrates the extracted use cases and relations in between them and the general categories. The dotted line in the middle indicates the separation between structural and behavioral verification tasks, with structural tasks on the left of the line and behavioral tasks on the right, respectively. As for the five categories, the classification into structural and behavioral verification tasks is not as strict, as behavioral tasks may be categorized

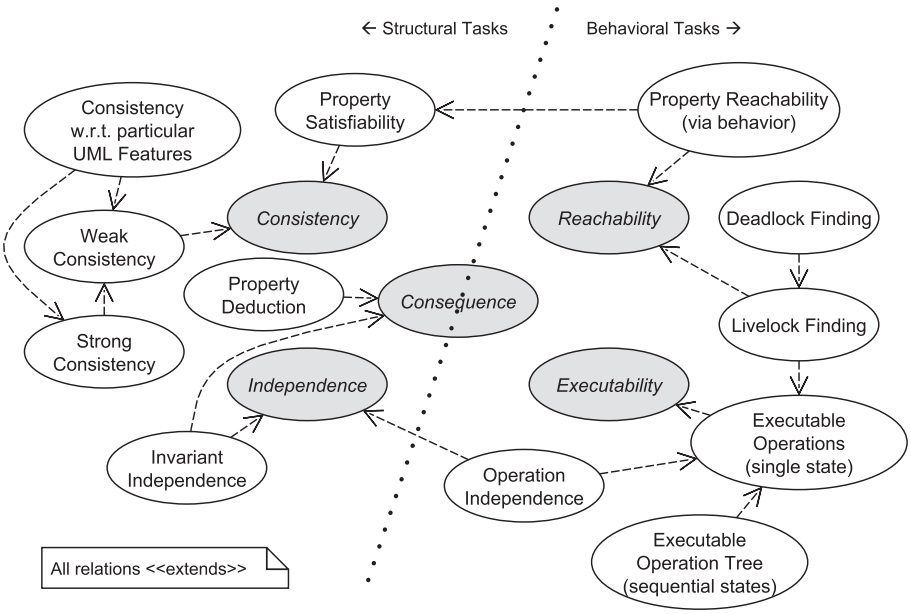


Fig. 2: Overview of verification task catalog.

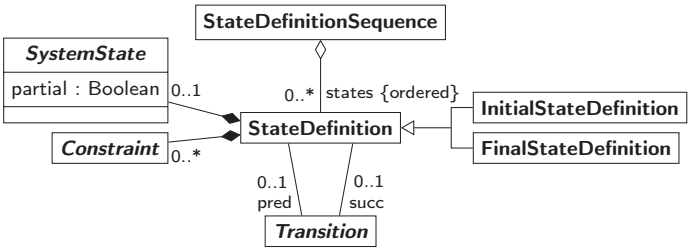


Fig. 3: Data model of inputs for verification tasks.

in a structural category (*Operation Independence*) or extend on structural tasks (*Property Reachability*). The listed verification tasks will be explained later in this section.

4.1 Verification Task Metamodel

In order to describe verification tasks in a formal fashion, we use the metamodel shown in Fig. 3. It shows an abstract metamodel that we use as a baseline for the declaration of the verification task input. The metamodel describes a structure that contains the information verification engines need to solve a certain verification task. The model creates a skeleton of information that must be filled in with a valid assignment by the verification engine.

Structural tasks utilize the `StateDefinition`, in the center of Fig. 3. This class represents a single abstract system state, which imposes no restrictions on a verification engine when generating a result. Using the abstract class `SystemState`, a concrete assignment

for this state can be given. The attribute `partial` determines, whether more elements may be added to this system state or not. Finally, the class `Constraint` allows to add (boolean) properties to a `StateDefinition` that have to be satisfied in the result. These methods to describe a system state can be mixed as necessary. To give a concrete example, a `StateDefinition` might be an object diagram or a state in a state machine.

While structural tasks only need the three classes mentioned above, behavioral tasks have access to additional information about the sequence in which these defined states occur (`StateDefinitionSequence`), the predecessors and successors of the states as well as the order and possibly the type of transitions between them. For example, the abstract class `Transition` might be extended to represent operation calls, state transitions in a state machine or signals. Finally, a `StateDefinition` can explicitly be declared as the initial or final state.

Figure 4 shows an example for a *Reachability* verification task pictured as an instance of the metamodel[2]. In the example, an initial and a final state is given by object diagrams. In the final state, both signals are set to green and the task is to find valid transitions from the initial to the final state, using the behavior defined in the model from Sect. 3. Since there is no path of transitions given in between the two system states, the amount and type of transitions is not restricted. Additionally, the system states could be further restricted by constraints in which the objects can be accessed using their names. The object names are also used to map them in between system states.

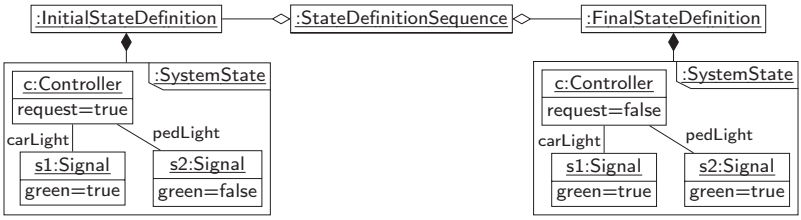


Fig. 4: Verification task metamodel example to define a reachability task with object diagrams.

4.2 Verification Tasks

In the following, the categories and their associated verification tasks from Fig. 2 are detailed and examples are given to illustrate the goals of selected tasks. The list of verification tasks, as framed in this work, is not meant to be complete. However, the provided list is a good viewpoint to show verification tasks that model checkers should be able to perform on UML/OCL models. We encourage others to extend the list and iteratively collect a more complete catalog of verification tasks.

4.2.1 Consistency

A *Consistency* verification task describes the instantiability property of a model, taking into account different sets of constraints applied, e.g., explicit and implicit model constraints, additional properties that serve as a certain verification goal, or even a reduced

[2] Due to space restrictions, we leave out exact details, how the abstract classes are extended to represent the information as an object diagram.

set of constraints. This category contains crucial verification tasks like showing whether a model contains contradictions and, therefore, might not be instantiable at all. Consistency problems are structural problems and do not involve behavior, like the execution of operations.

Weak Consistency This task describes the general instantiability of a model. The goal is to generate a system state that uses at least some model elements while satisfying all model constraints. Note that invariants assigned to classes that are not instantiated, are satisfied by design in the standards.

Strong Consistency This task is an extension of *Weak Consistency* by the property that *all* model elements have to be considered in the generated system state, i.e., at least one object of all classes and one link of all associations have to be instantiated.

Consistency w.r.t. particular UML Features This task extends the former two tasks allowing particular UML features, such as multiplicities, aggregation and composition rules or invariants, to be ignored.

Property Satisfiability This task represents a consistency task including external properties in addition to the model. The goal is to find a valid system state satisfying all model constraints plus the additional properties. Figure 5 shows a small example requiring at least a system state with a controller `c`. In addition, a specific property is specified as an OCL constraint, requiring both signals of this controller show the green signal, which fails due to the invariant `safety`.

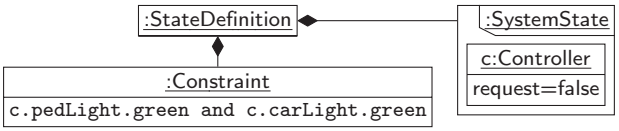


Fig. 5: Verification task model for property satisfiability task.

4.2.2 Independence

Independence describes verification tasks that reason about (in)dependencies between model elements. This includes any type of dependencies that can exist between, e.g., attributes, roles or invariants. In addition, tasks setting these dependencies in relation also belong in this category.

Invariant Independency The goal of this task is to check whether invariants exist that are implicitly specified by one or more others and are therefore always satisfied when the dependant invariants are satisfied. Further extensions of this task is the identification of which invariants imply the dependent invariant.

4.2.3 Consequence

Verification tasks in the category *Consequence* describe tasks that deduct information from a model. These consequences are inherent in the model and are given by the model constraints, e.g., multiplicities, invariants or more complex deductions.

Property Deduction This task describes the action of identifying information not explicitly in a model, but that are clearly implied by one or more model elements from

the model. In contrast to the *Property Satisfiability* task, these information deducted from the model are not restricted to boolean properties.

4.2.4 Reachability

Reachability verification tasks include all tasks with a certain defined goal in mind that is reached by executing the behavior of a model such as operations or state machine state transitions. In contrast to *Consistency* verification tasks, *Reachability* tasks involve at least two system states that are connected by model transitions, defined by the behavior of the model.

Property Reachability Similar to the *Property Satisfiability* verification task, this task checks the satisfiability of properties in a model. This tasks, however, tries to satisfy them by executing model transitions and additionally allows to specify initial and intermediate system states that must be included in the simulation. The properties to be reached can be given as system states or constraints, as defined in the metamodel in Fig. 3. In the running example, it is desirable to reach a state where the pedestrians are finally allowed to cross the street, when the signals are currently allowing the cars to cross, and vice versa.

4.2.5 Executability

In the *Executability* category are all verification tasks that focus on the transitions and their contracts between system states. While this paper focuses on operation calls, the metamodel in Sect. 4.1 allows for any form of state transition.

Livelock Finding This task identifies state transitions that result in a *livelock*, i.e., the system is in a state where there is no possible sequence of transitions to reach a defined end state, while transitions can still be executed.

Deadlock Finding This verification task is the extension of the *Livelock* task, searching for reachable system states, where the system comes to a complete halt and no further transition is possible, without being in a defined end state. This task can make sure that, in the running example, there is no possibility to get into a state where only either cars or pedestrians are allowed to cross the street (forever).

Executable Operations The goal of this verification task is to identify all executable transitions of a single system state. In the running example this is achieved by evaluating the preconditions of all operations against the given system state. This task is the basis for many other verification tasks.

Executable Operation Tree This verification task extends the previous task by checking the possible transitions not only for a single system state, but also simulating the operation calls and iteratively evaluate the executable state transitions, building a full tree of operation call sequences up to a given depth. Again the task can be restricted by giving a certain final state as a goal or constraining the transition sequence using simple OCL. The constraints on the sequence can be as complex as temporal logic.

Operation Independence This task introduces the detection of dependencies in operations from the former verification task. An example is the identification of mutually exclusive operation calls, i.e., identifying operation calls that are never available at the same time in a single system state.

5 Conclusion

We have presented a catalog of general verification tasks for UML/OCL verification tasks including their categorization into five groups. All tasks are individually detailed to establish a fine granular differentiation between their goals. These definitions shall help modellers to communicate with each other and unify the term usage in verification engines. In addition, we have presented a metamodel to represent these verification tasks in a formal fashion.

References

[An07] Anastasakis, Kyriakos; Bordbar, Behzad; Georg, Geri; Ray, Indrakshi: UML2Alloy: A Challenging Model Transformation. In: MoDELS. Springer, pp. 436–450, 2007.

[Ba12] Banerjee, Ansuman; Ray, Sayak; Dasgupta, Pallab; Chakrabarti, P. P.; Ramesh, S.; Ganesan, P. Vignesh V.: A dynamic assertion-based verification platform for validation of UML designs. ACM SIGSOFT Software Engineering Notes, 37(1):1–14, 2012.

[Be15] Benduhn, Fabian; Thüm, Thomas; Lochau, Malte; Leich, Thomas; Saake, Gunter: A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems. VaMoS '15. ACM, pp. 80:80–80:87, 2015.

[CCR08] Cabot, Jordi; Clarisó, Robert; Riera, Daniel: Verification of UML/OCL Class Diagrams using Constraint Programming. In: First International Conference on Software Testing Verification and Validation, ICST 2008. IEEE Computer Society, pp. 73–80, 2008.

[CKZ11] Choppy, Christine; Klai, Kais; Zidani, Hacene: Formal Verification of UML State Diagrams: A Petri Net based Approach. Softw. Eng. Notes, 36(1):1–8, 2011.

[CS13] Calegari, Daniel; Szasz, Nora: Verification of Model Transformations: A Survey of the State-of-the-Art. Electronic Notes in Theoretical Computer Science, 292:5 – 25, 2013. Proceedings of the XXXVIII Latin American Conference in Informatics (CLEI).

[EW04] Eshuis, Rik; Wieringa, Roel: Tool Support for Verifying UML Activity Diagrams. ITSE, 30(7):437–447, 2004.

[GHH14] Gogolla, Martin; Hamann, Lars; Hilken, Frank: Checking Transformation Model Properties with a UML and OCL Model Validator. In (Amrani, Moussa; Syriani, Eugene; Wimmer, Manuel, eds): Proc. 3rd Int. Workshop on Verification of Model Transformation (VOLT'2014). CEUR Proceedings, Vol. 1325, pp. 16–25, 2014.

[GKH09] Gogolla, Martin; Kuhlmann, Mirco; Hamann, Lars: Consistency, Independence and Consequences in UML and OCL Models. In (Dubois, Catherine, ed.): Tests and Proofs, Third International Conference, TAP. volume 5668 of LNCS. Springer, pp. 90–104, 2009.

[La07] Lam, Vitus S. W.: A Formalism for Reasoning about UML Activity Diagrams. Nordic Jrnl. of Comp., 14(1):43–64, 2007.

[Ro14] Rodríguez, Ricardo J.; Fredlund, Lars-Åke; Herranz-Nieva, Ángel; Mariño, Julio: Execution and Verification of UML State Machines with Erlang. In: Software Engineering and Formal Methods. pp. 284–289, 2014.

[SWD11] Soeken, Mathias; Wille, Robert; Drechsler, Rolf: Verifying Dynamic Aspects of UML Models. In: DATE. IEEE, pp. 1077–1082, 2011.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 125

Tool Support for Model Transformations: On Solutions using Internal Languages

Georg Hinkel[1] and Thomas Goldschmidt[2]

Abstract: Model-driven engineering (MDE) has proven to be a useful approach to cope with todays ever growing complexity in the development of software systems, yet it is not widely applied in industry. As suggested by multiple studies, tool support is a major factor for this lack of adoption. Existing tools for MDE, in particular model transformation approaches, are often developed by small teams and cannot keep up with advanced tool support for mainstream languages such as provided by IntelliJ or Visual Studio. In this paper, we propose an approach to leverage existing tool support for model transformation using internal model transformation languages and investigate design decisions and their consequences for inherited tool support. The findings are used for the design of an internal model transformation language on the .NET platform.

Keywords: Model-driven Engineering, Model Transformation, Internal DSL, C#

1 Introduction

While in the past, increasing complexity of software systems has been tackled by an increasing abstraction of the programming language, it seems like the abstraction level of modern programming languages can hardly be raised without losing general purpose applicability. Therefore, in recent years, many domain-specific languages (DSLs) [Fow10] have been proposed that offer a raised abstraction level at the price of limited expressiveness.

The usage of such domain-specific language requires a specification how these languages are executed. A popular approach for this is to map a domain-specific language to a target platform by a transformation. Since such a transformation determines the execution semantics of the source languages instances, model transformations are sometimes called the 'heart and soul' of model-driven approaches that should be supported by dedicated languages [SK03].

This has lead to a variety of model transformation approaches [CH06] incorporating high-level abstractions like the composition of model transformations into rules describing the transformation for a particular model element. While these languages produce more concise, more understandable and sometimes even more performant (cf. eg. [GR13]) model transformations than general purpose languages, the model-driven approach is still not

[1] FZI Forschungszentrum Informatik, Software Engineering Division, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, hinkel@fzi.de
[2] ABB Corporate Research, Software Systems, Wallstadter Straße 59, 68526 Ladenburg, thomas.goldschmidt@de.abb.com

widely adopted in industry and the question is why. Multiple studies [Sta06; Moh+09; Whi+13] suggest that a major factor in this decision is the tool support, particularly also of model transformations given their importance in model-driven approaches. Recent studies [Moh+13] suggest that the tool support is still not satisfactory.

But as the model-driven approach is not widely adopted, relatively few resources are spent to improve the tools, at least in comparison to IDEs for mainstream languages like IntelliJ or Visual Studio. These tools have a massive user base and thus much more resources are spent for the improvement of the tools. Furthermore, many model transformation tools are maintained by researchers with few incentives to implement in principle long-known tool concepts in their model transformation tools, simply because of the lack of insights generated by these mostly laborious tasks.

Furthermore, as Meyerovich suggests [MR13], many developers do not appreciate to switch their primary programming languages and do so only if there is a significant amount of code they can reuse or if management requires them to do so. This is reasonable since such a change often makes valuable knowledge of particular technologies superfluous. Furthermore, similar concepts are sometimes implemented in slightly different ways, causing subtle bugs. As an example in the world of model transformations, the difference between *is-kind-of* and *is-type-of* in OCL often causes confusions for developers not confronted with it on an everyday-basis.

A promising approach to tackle this problem of 1. general purpose languages with the lack of model transformation concepts on the one side and 2. dedicated model transformation languages with lacking tool support on the other is to combine both worlds using an internal DSL. To gain best tool support, the model transformation language should be hosted in a mainstream general-purpose language such as Java or C#. This allows to combine high-level abstractions for model transformations with advanced tool support.

However, so far internal transformation languages do not exist for all often used general-purpose languages. This raises the problem how to design an internal transformation for a language so far not covered and how to implement this transformation DSL with respect to tool support.

In this paper, we tackle this problem as we extract our experience with the design of the NMF Transformation Language (NTL)[Hin13] regarding design for tool support reuse. We discuss the design alternatives how to map model transformation concepts (in particular transformation rules) to code artifacts in an internal DSL and explain limitations and consequences. We then describe briefly how these design decisions are implemented in NTL.

In the remainder of this paper, we first present related work in Section 2. Section 3 discusses how transformation rules can be embedded in general-purpose object-oriented programming concepts aiming for optimal tool support reuse. Section 4 explains our implementation in the internal model transformation language NTL. Finally, we conclude the paper in Section 5.

2 Related Work

There exists a variety of model transformation approaches, surveyed and summarized for example by Czarnecki et al. [CH06]. In the remainder of this section, we concentrate on those implemented as internal DSLs.

A language that has been used as host language for model transformation several times is Scala [GWS12; KCF14] . Scala has not as many users as Java or C# but still advanced tool support is available. However, the language adoption problem remains, i.e. fewer developers know Scala than Java or C#.

Surprisingly, the most often used mainstream languages to the best of our knowledge have hardly been used as a host language for model transformation yet. Next to NTL and its close relative NMF Synchronizations [Hin15], we are only aware of two approaches using Java [TL12] which is rather focussed on pattern matching and SDMLib [Zün+13], an internal DSL for the Fujaba tool using a method chaining syntax.

In this paper, we discuss how an internal model transformation language can be constructed specifically for the inheritance of tool support, but DSLs have been used in a number of approaches for a multitude of different rationales, including the ease of development [BH11], type safety [GWS12] or language adoption [Hin+15].

3 Implementing Transformation Rules with respect to Tool Support

This section discusses the implications of different ways to represent transformation rules in the object-oriented design aiming specifically to gain optimal tool support for model transformations. Our discussion is based on strongly-typed multi-paradigm programming languages such as Java or C# as host languages where there is rich tool support available.

3.1 Editing Support

According to a study on the usage of the Eclipse IDE with 41 Java developers, the basic editing operations like delete, copy, cut and paste are upon the most commonly used editing commands [MKF06]. These commands are supported by any editor. However, the study showed that also more sophisticated tools were often used, indicating that they raise productivity. The most often used tool support beyond the basic commands was code completion that was used by every developer, making up in the average 6.7% of all executed commands. This study was made in 2006 and code completion has been improved by learning from examples, frequency or mined associations [RL08; BMM09]. While many model transformation tools support a basic code completion listing all available members in alphabetic order, more advanced code completion is usually limited to large mainstream IDEs.

Code completion requires strongly typed environments since in this case the tool knows what methods are available for a given object. Thus, the signature of a transformation rule

must be known to the compiler. This can be achieved either when the transformation rule is represented by a method or by turning the transformation rule signature into generic type parameter. The problem with the representation as a method is that it is very hard to decide when to create a trace entry. Transformation rules like ATL, ETL and QVT-O solve this problem by dividing the transformation rule execution into phases. In SIGMA which represents transformation rules as methods, the problem is solved by allowing only a single transformation phase specified by the user.

The approach of turning a transformation rule signature into generic type parameters is more flexible, but there are multiple possible implementations. The difference between these implementations is the artifact that represents a given transformation rule. Generic type parameters can be created for methods or classes. In case of generic parameters of methods, each transformation rule would be a call of a generic transformation rule method that creates the transformation rule, taking in additional configuration such as different phases of the transformation.

```
var state2place = TransformationRule<State, Place>(
  createOutput: (state, context) => ...,
  transform: (state, place, context) => ...
);
```

List. 1: Representing transformation rules as method calls

An example how the transformation of states of a finite state machine to places of a Petri net looks like when transformation rules are implemented as method calls is shown in Listing 1. In this listing, we assume an optional `context` parameter that can be used for tracing purposes. The example uses named parameters which are not available in all languages (and usually optional where they are available). Other options include method chaining syntaxes. An example of these languages is SCALAMTL, as method chains have a suitable syntax in Scala.

The representation as inheritance means that there is a generic transformation rule class that is inherited from for each transformation rule, passing the type signature again as generic type parameters. Here, different phases of a transformation rule can easily be represented by different methods of the class which the transformation rule has to override.

```
class State2Place : TransformationRule<State, Place> {
  Place CreateOutput(State state, Context context) { ... }
  void Transform(State state, Place place, Context context) {...}
}
```

List. 2: Representing transformation rules as classes

The representation of transformation rules as classes with a certain inheritance relation is sketched in Listing 2. Here, the concept of a transformation rule is implemented in a generic class which is inherited from. The transformation phases are represented as overridden methods.

The design decision whether to implement transformation rules as method calls or as classes has several important consequences. While the syntax of the method calls contains

less syntactic boilerplate and is thus more concise in terms of lines of code, the latter version using inheritance has the important advantage that being types, transformation rules are reflected in metadata. This has advantages for visualization as we will discuss in the next section. Furthermore, an object as the result of a method call can only be referenced once it has been created while a class can be referenced regardless of the order in the code (in most languages). This is problematic when the abstract syntax (cf. Fig. **??**) contains a cross-reference to transformation rules, when traces are explicit. Thus, for example in SCALAMTL, the traces are implicit and cannot be made explicit.

3.2 Navigation Support

A large proportion of development activities is devoted on the analysis of existing code and navigation through it [MKF06]. However, the navigation support of the mostly used search commands (searching for references to a selected element or its definition) can be derived independently of the transformation rule representation, if there is a representation as a code element at all (as opposed e.g. to pure naming conventions).

However, Rentschler et al. have shown that a visualization of a model transformations structure aids the navigation and is thus helpful for the maintainability of model transformations [Ren+13]. A similar visualization is getting common for general-purpose object-oriented code visualizing the usage of members within a class or the usage of classes within a package, as for example with Code Maps in Visual Studio. This analysis is based on metadata, i.e. classes, methods and their interrelations based on the methods' bodies. Objects as results of method calls are not part of this metadata since they are runtime artifacts and cannot be predicted at compile-time in general. As a consequence, code visualizations based on this metadata is not available when transformation rules were represented in method calls.

For internal languages, inherited visualization is of particular importance. Unlike external languages, they are merely guidelines how to use a framework but these guidelines are not enforced by the host language compiler[3]. As a consequence, static analysis like visualization specifically created for the internal language is of limited applicability. For analyses that look at the big picture like visualizations of the entire transformation, this means that it is hard to create something above the inherited visualization support. Alternatively, dynamic visualization as supported by SDMLIB are a viable approach, but are hard to integrate in the development environment and laborious to develop.

For the implementation of transformation rules as types as outlined above, the usage of a transformation rule is the same as the usage of this type. Therefore, visualization techniques to show dependencies in object-oriented programming can be used to visualize the structure of model transformations. An implementation of transformation rules as method cancels the possibility of inherited visualizations.

[3] Technologies like the modular compiler Roslyn give internal languages the chance to enrich the host language compiler

4 The NMF Transformations Language (NTL)

This section describes the NMF Transformations Language (NTL), the concrete syntax of the model transformation framework of NMF. The language uses C# as host language and is able to describe model transformations from and to arbitrary runtime objects. The implementation is part of NMF and thus available as open-source[4].

As we are aiming for a comprehensive transformation language, we represent transformation rules as classes that inherit from a common generic transformation rule class and pass in their signature as generic type parameters (cf. Section 3). The different phases of the transformation rule (cf. Section **??**) are specified by overriding methods from the base class. Thus, transformation rules in NTL exactly look like sketched in Listing 2 except for some syntactic boilerplate: In C#, overriding methods must repeat the entire signature of the base method accompanied by the `override` keyword.

```
1 public class FSM2PN : ReflectiveTransformation {
2   public class State2Place : TransformationRule<State, Place> { ... }
3 }
```

List. 3: A transformation in NTL

The assembly of a model transformation of transformation rules is done by adding the transformation rule as public nested types of the transformation class which in turns inherit from `ReflectiveTransformation`. An example is presented in Listing 3. As a consequence, the declaration of the transformation rule and its registration with the transformation coincide, decreasing maintenance efforts.

```
1 public class State2Place : TransformationRule<State, Place> {
2   protected override RegisterDependencies() {
3     CallMany(Rule<Transition2Transition>(),
4       selector: s => s.Outgoing,
5       persistor: (p,transitions) => p.From.AddRange(transitions));
6   }
7 }
```

List. 4: Specifying dependencies in NTL

The specification of dependencies for a transformation rule is sketched in Listing 4. Dependencies are created in a dedicated method `RegisterDependencies` (see line 2) which is run by the transformation engine at initialization of the transformation.

Unlike transformation rules, dependencies themselves are merely uninteresting in code visualization. Furthermore, dependencies need not to be cross-referenced. Therefore, we represent them as method calls in a dedicated function of transformation rules. The attributes of dependency elements of the abstract syntax are simply passed as method call arguments. In line 3, such a call is made creating a dependency of the *State2Place* rule to *Transition2Transition* for each outgoing transition of a state in the state machine (line 4). The resulting transitions are then added to the *From* reference of the Petri net place which is the transformation result of the current state (line 5).

[4] `http://nmf.codeplex.com`

The entire finite state machine to Petri nets transformation can be found in [Hin13]. We abbreviate it here for space limitations.

5 Conclusion

Despite the improvements in terms of productivity, Model-driven engineering still lacks an industry adoption. In this paper, we have proposed an approach how this tool support problem can be solved by internal model transformation languages by exploring the design alternatives how model transformation rules can be represented in object-oriented design. We have shown how this discussion has lead to the development of NTL.

There is no unique way of implementing transformation rules in an object-oriented language. The implementation choices are trade-off decisions. An implementations of transformation rules as methods or method calls lead to a more concise syntax with less syntactic boilerplate but yield restrictions. Implementations as methods restrict the transformation language to a single operational phase in transformation rules and method calls make explicit tracing hard and cancel inherited visualizations based on metadata. An implementation alternative without these shortcomings at the price of a less concise language is the implementation as classes inheriting from a common transformation rule class.

References

[BH11] H. Barringer and K. Havelund. *TraceContract: A Scala DSL for trace analysis*. Springer, 2011.

[BMM09] M. Bruch, M. Monperrus, and M. Mezini. "Learning from examples to improve code completion systems". In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 213–222.

[CH06] K. Czarnecki and S. Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–645.

[Fow10] M. Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

[GR13] P. V. Gorp and L. Rose. "The Petri-Nets to Statecharts Transformation Case". In: *Sixth Transformation Tool Contest (TTC 2013)*. EPTCS. 2013.

[GWS12] L. George, A. Wider, and M. Scheidgen. "Type-Safe model transformation languages as internal DSLs in scala". In: *Theory and Practice of Model Transformations*. Springer, 2012, pp. 160–175.

[Hin+15] G. Hinkel et al. "A Domain-Specific Language (DSL) for Integrating Neuronal Networks in Robot Control". In: *2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. 2015.

[Hin13] G. Hinkel. *An approach to maintainable model transformations using internal DSLs*. Master thesis, Karlsruhe Institute of Technology. 2013.

[Hin15] G. Hinkel. “Change Propagation in an Internal Model Transformation Language”. In: *Theory and Practice of Model Transformations*. Springer, 2015, pp. 3–17.

[KCF14] F. Křikava, P. Collet, and R. B. France. “SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations”. In: *Model-Driven Engineering Languages and Systems*. Springer, 2014, pp. 569–585.

[MKF06] G. C. Murphy, M. Kersten, and L. Findlater. “How are Java software developers using the Elipse IDE?” In: *Software, IEEE* 23.4 (2006), pp. 76–83.

[Moh+09] P. Mohagheghi et al. “MDE adoption in industry: challenges and success criteria”. In: *Models in Software Engineering*. Springer, 2009, pp. 54–59.

[Moh+13] P. Mohagheghi et al. “An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases”. In: *Empirical Software Engineering* 18.1 (2013), pp. 89–116.

[MR13] L. A. Meyerovich and A. S. Rabkin. “Empirical analysis of programming language adoption”. In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. ACM. 2013, pp. 1–18.

[Ren+13] A. Rentschler et al. “Interactive visual analytics for efficient maintenance of model transformations”. In: *Theory and Practice of Model Transformations*. Springer, 2013, pp. 141–157.

[RL08] R. Robbes and M. Lanza. “How program history can improve code completion”. In: *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE. 2008, pp. 317–326.

[SK03] S. Sendall and W. Kozaczynski. “Model transformation: The heart and soul of model-driven software development”. In: *Software, IEEE* 20.5 (2003), pp. 42–45.

[Sta06] M. Staron. “Adopting model driven software development in industry–a case study at two companies”. In: *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 57–72.

[TL12] B. Trancón y Widemann and M. Lepper. “Paisley: pattern matching à la carte”. In: *Theory and Practice of Model Transformations*. Springer, 2012, pp. 240–247.

[Whi+13] J. Whittle et al. “Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?” In: *Model-Driven Engineering Languages and Systems*. Vol. 8107. LNCS. Springer Berlin Heidelberg, 2013, pp. 1–17.

[Zün+13] A. Zündorf et al. “Story Driven Modeling Libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet-Statechart case”. In: *Sixth Transformation Tool Contest (TTC 2013), ser. EPTCS* (2013).

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 133

An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information

Pedram Mir Seyed Nazari[1], Alexander Roth[1], Bernhard Rumpe[1]

Abstract: Code generation is regarded as an essential part of model-driven development (MDD) to systematically transform the abstract models to concrete code. One current challenges of template-based code generation is that output-specific information, i.e., information about the generated source code, is not explicitly modeled and, thus, not accessible during code generation. Existing approaches try to either parse the generated output or store it in a data structure before writing into a file. In this paper, we propose a first approach to explicitly model parts of the generated output. These modeled parts are stored in a symbol for efficient management. During code generation this information can be accessed to ensure that the composition of the overall generated source code is valid. We achieve this goal by creating a domain model of relevant generator output information, extending the symbol table to store this information, and adapt the overall code generation process.

Keywords: Symbol Table, Output-Specific Generator Information, Code Generation

1 Introduction

In model-driven development (MDD) code generation is an essential part to systematically generate detailed code from abstract input models. To bridge the gap between problem domain (abstract models) and solution domain (concrete code), MDD lifts the input models to primary artifacts in the development process. Regardless of the importance, code generator development is still a labor-intense and time-consuming task, where approaches to explicitly manage output-specific information are still lacking.

Explicitly management of output-specific code generator information, i.e., information about the generated source code, is essential for code generation to ensure that the generated source code is valid, i.e., well-formed. More importantly, it is necessary in the development process of code generators to split development tasks and in the maintenance phase as a documentation. For example, consider Java code is generated from UML class diagrams [Ru11]. In order to access parts of those generated Java classes are to be accessed during code generator runtime, the information of the relation of class diagram elements to Java elements is required such as class instantiation via the factory pattern [Ga95] versus direct instantiation via `new`-constructs.

Current code generator frameworks, e.g., [Me15, Xt15, Ac15, Je15], primarily focus on the code generation process and the development of code generators but mainly neglect explicit modeling of code generator output. Moreover, round-trip engineering [MER99]

[1] RWTH Aachen University, Software Engineering, http://www.se-rwth.de

and reverse engineering [CC90] try to recreate models from the generated output. An approach to explicitly model information, which is exchanged during code generation, has been presented [JMS08]. However, the output still needs to either be first generated or approaches to address the required parts of the generated source code are lacking.

Hence, in this paper we present our approach to make output-specific code generator information explicit. As this information is dependent on the used input and output language, we present a preliminary domain model for a code generator that uses a variant of UML class diagrams [OM15] as input and Java as output. In this domain model, we use existing approaches [Ru11, Ru12] to map elements of UML class diagrams to Java code and provide an extension to manage Java object instantiation and field access via accessors and mutators. By making such information explicit, we enable code generator developers to exchange this information during development. Moreover, we make this information accessible at generation-time by extending the symbol table and the code generation process to allow storing arbitrary information.

Hence, we first introduce the basic concepts of symbol tables and code generation used in the MontiCore framework in Section 2. Then, we present our approach to manage output-specific code generator information by extending the symbol table and the code generation process (Section 3). Finally, we conclude our paper in Section 5.

2 Symbol Table and Code Generation

As the foundation for all aspects of language definition, language processing, and template-based code generation, we use the MontiCore language workbench [KRV10]. It uses a grammar defining the language to be processed and generates a parser and infrastructure for language processing based on this grammar. The generated infrastructure can be used to parse models conforming the defined grammar.

During processing of models the parser creates a *abstract syntax tree (AST)*, (an internal representation of the input model). This abstract representation is used for further phases of language processing, e.g., context condition checking and code generation. In addition, a *symbol table* is created in order to store relevant information for each model element.

2.1 Symbol Table

A symbol table (ST) is a data structure that maps names to associated model element information. In MontiCore, a *symbol* is an entry in the ST and represents a (named) model element [HNR15]. It contains all essential information related to that element. Different kinds of model elements, e.g., method and field in Java, are distinguished by corresponding *symbol kinds*. The main purpose of the symbol table is an efficient finding of model element specific information such as its type and its signature.

Compared to classical symbol tables, which are typically simple hash tables [Ah07], the symbol table in MontiCore is a combination of a (conceptual) table and the semantic model as described in [Fo10]. Its underlying infrastructure is a scope-tree containing a *collection*

of symbols (cf. [Pa10]). Furthermore, it serves as a language-unspecific infrastructure for an efficient and effective integration of heterogeneous modeling languages [Lo13, Ha15])

Besides the information defined in the model element and represented by a symbol in the symbol table, a symbol can also contain information that is not explicitly part of that model element. For example, a Java field symbol can state whether it shadows a field of the super class or not. In addition, the source position of the model element can be stored in the symbol. Both information are not explicitly stated in the model element, but can be managed by the symbol. This allows to associate any kind of information—even technical information such as the source position— with the corresponding model element. We have even shown that a symbol table can manage code generator customizations [NRR15].

2.2 Code Generation

The code generation process of MontiCore is a mix of template-based and transformation-based code generation as shown in Figure 1. After the parser has created the AST, multiple transformations can be applied to transform the AST by adding, removing, or changing elements of the AST. The overall goal of the transformations is to make it fit the needed AST for code generation. During the transformation steps templates can be attached to AST elements in order to explicitly define the template to be used for this particular element. Certainly, this approach has limitations when generating non object-oriented code or when the input model is not a structural description that can be used for code generation. Thus, in the remainder of this paper we focus on a modeling language that is a variant of UML class diagrams [OM15] and Java as the output language of the code generator.

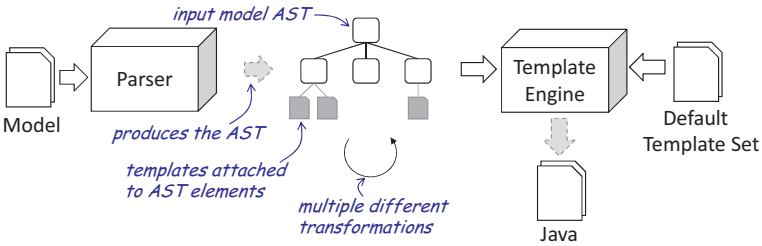


Fig. 1: Overview of a template-based and transformation-based code generation process.

After the transformations have been successfully applied, the AST is passed to the Template Engine. In addition, a default set of templates, which describe how to generate Java code from the input model, is passed to the template engine. When the template engine is started, it traverses the input AST and for each element executes either the attached templates of one of the default templates depending on the type of the AST element. Finally, the generated output is written to a file.

3 Managing Output-Specific Information with the Symbol Table

Our presented approach to manage output-specific code generator information is based on three elements. First, a common understanding of output-specific code generator information is needed. In general, this information is concerned with output language specific

elements and concepts, e.g., object instantiation in Java. Second, an extension to the symbol table is required in order to add output-specific information and make it available at generation-time, i.e., run-time of the code generator. Third, the code generation approach needs to be adapted such that the information is added to the symbol table. Subsequently, we elaborate on each of the three main steps in more detail.

3.1 A Preliminary Domain Model for Class Diagrams and Java

A domain model of code generator output specific information depends on the output of the code generator and the input language. Hence, aiming for a general domain model for code generator output specific information is challenging and possibly not feasible. However, restricting the input language to UML class diagrams and the output language to Java, we try to provide a preliminary domain model that shows how code generator output-specific information can be modeled and managed with a symbol table. We do not claim for completeness of the domain model. Instead, we try to give an idea of how to model code generator output-specific information.

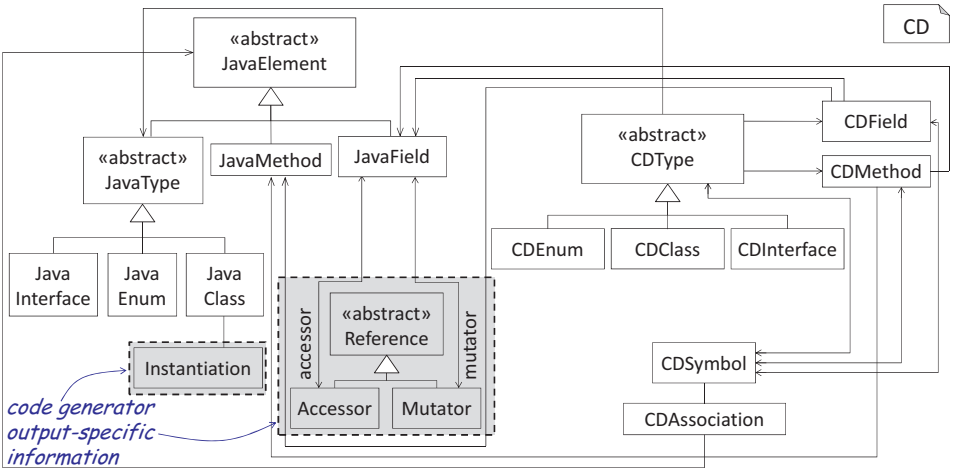


Fig. 2: Mapping of Symbols to Java Symbols and additional Generator Information

Our domain model in Figure 2 shows how UML class diagram symbols are mapped to Java symbols based on [Ru11, Ru12]. In this domain model, a `CDType`, which may represent a UML enum, interface or class, is mapped to a `JavaType`. We do not restrict the mapping to `JavaClasses`, because it may be necessary to generate interfaces or a modeled class. Moreover, each `CDField` is mapped to `JavaMethod` and `JavaField`. The mapping of a `CDField` to `JavaMethods` is optional as accessors and mutators may not be wanted. A `CDMethods` symbol is mapped to `JavaMethod` and `JavaField`. An example for a UML method that is mapped to a `JavaField` is an accessor that is mapped to the generated `JavaField` to allow for direct variable access.

Figure 2 gives an example for code generator output-specific information. This information is relevant for code generator developers and should be accessible during generation time rather than after code generation. Focusing on our small example, we have identified two

types of code generator output-specific information. First, a `JavaField`, when mapped to Java code, can have `Accessors` and `Mutators`. This information is relevant during code generation as the generated code should access the field using the generated accessor and mutator. Thus, this information should be modeled explicitly and be accessible before the code is generated. Moreover, for `JavaClasses` the information relevant for creating instances of this class is required, e.g using the Singleton pattern [Ga95].

3.2 An Extension to the Symbol Table

Having an understanding of the mapping and additional information to be stored in the symbol table, we extend MontiCore's symbol table infrastructure to efficiently manage this information. The subsequent description is reduced to the essential parts and mainly focuses on the extensions, as shown in Figure 3. In [HNR15], we introduce the symbol table infrastructure in detail.

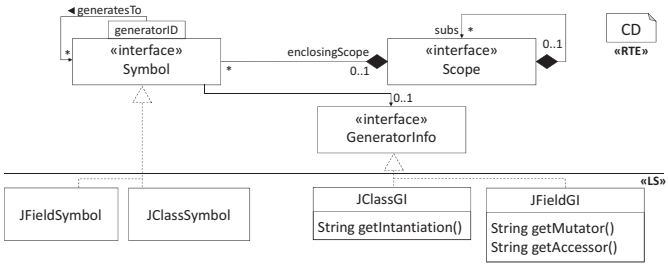


Fig. 3: Extended Symbol Table Infrastructure

As a first step, we enriched a symbol with information about the symbols of the target language it generates to. Moreover, symbols of the source language are associated with the corresponding symbols of the target language. For example, a class diagram field (source) can be generated as a Java field (target). Hence, the `CDFieldSymbol` maps to the corresponding `JavaFieldSymbol` (see Sect. 3). However, different generators can lead to different mappings and, thus, a unique generator id is used for each generator.

Second, each symbol now can optionally store generator-specific information, represented by the `GeneratorInfo` interface. `GeneratorInfo` must be implemented for each symbol of a target language and provide the required information. In the example of Java as the target language, information for, among others, classes and fields is needed, which are presented by `JavaClassGI` and `JavaFieldGI`, respectively. The former provides information such as how the generated class is instantiated, while the latter states how the field of the generated class is changed or accessed.

3.3 An Extension to Code Generation to handle Output-Specific Information

All modeled output-specific information is added to the symbol table to make it available at generation-time. Two different approaches can be used to add this information. First, before generation-time the transformations and templates can be parsed and the required information can be extracted. To identify relevant information comments or keywords may

be used. While this approach makes sure that all information is available before generation-time, it has the disadvantage of processing all transformations and templates. In consequence, a supporting infrastructure to parse templates and transformations is necessary.

The second approach for adding all relevant information to the symbol table is to add it at generation-time. In particular, this means that all output-specific information is added to the symbol table while the code generation process is running. The information is not available before generation-time but still available at generation-time. A benefit of this approach is that no parsing of transformations and templates is required and the provided infrastructure can be kept small by providing an API to add this information. For our example in Figure 2, we can to provide the following methods:

- `toJavaType(CDType s, String className)`: Defines that a `CDType` is mapped to a `JavaType` with the name `className`.
- `toJavaField(CDField s, String fieldName)`: A mapping `CDField` to a `JavaField` is stored in the symbol table with the `fieldName` as the name of the generated Java field.
- `toJavaMethod(CDMethod s, String methodName)`: To define a mapping of a `CDMethod` to a `JavaMethod` the symbol table creates a Java method with the name `methodName`.
- `addInstantiation(JavaClass c, String code)`: In order to explicitly model object instantiation and store it in the symbol table, the API allows to add piece of `code` of type `String` to a Java class. For instance, to regard the Factory pattern, the piece of code can be "BookFactory.create()" for the Java class `Book`.
- `setAccessor(JavaField, String code)`: A mutator for a `JavaField` can be defined as a piece of `code` that represents, e.g., the name of the method ("getTitle" for a field named "title").
- `setMutator(JavaField, String code)`: For mutators the method is the same as for accessors. Additionally, we assume that each mutator requires one argument. Hence, when accessing this information in the symbol table a parameter should be passed. This is used to create the resulting string for the mutator.

A disadvantage is that the transformations and code templates have an execution order in which they have to be executed. If the execution order is violated, the information may not be available. In other words, the symbol in the symbol table cannot be resolved.

4 Related Work

Explicit modeling of output-specific code generator information is, to our knowledge, only hardly addressed by current literature. A closely related approach has been presented in [JMS08]. Here, a code generator is explicitly modeled via small interconnected services, which exchange information at runtime. This approach is similar to our presented approach as the exchanged information between serviced may contain generated informa-

tion. In contrast, our presented approach proposes explicit modeling of this information and efficient management by using a symbol table.

Another approach that can be used to exchange information about the generate output has been presented by [ZR11]. The authors propose to generate the source code into containers before writing it into files. Hence, the complete source code is available at generation-time. However, as the authors are mainly concerned with producing syntactically correct output, there is no approach to address parts of the generated code as proposed by this paper. This is, however, essential to address composition of the generated source code, e.g., instantiation of generated Java classes.

Finally, an extension to round-trip engineering has been proposed to address the framework-provided abstractions via a dedicated domain-specific language (DSL) [AC06]. Rather than proposing a DSL, we explicitly model output-specific information using UML class diagrams and additionally provide efficient management at generation-time.

5 Conclusion

As code generation is regarded as an essential part of model-driven development to generate source code, output-specific code generator information has to be regarded in order to generate valid source code and decompose the generator development. In this paper, we presented a first approach to make output-specific code generator information explicit.

Our proposed approach consists of three steps. First, the relevant information is collected in a domain model. Based on this domain model the symbol table is extended to manage this information. Using the symbol table as an infrastructure has the benefit that the management is more efficient and no additional infrastructure is required. Finally, in the last step the code generation process needs to be adapted in order to make use of the stored information. We have applied this approach to a small use case to show how to model output-specific information for a UML class diagram to Java code generator. In particular, we focused on information related to object instantiation, and mutaturs and accessors for fields. In future, we plan to extend this approach to more real world examples.

References

[AC06] Antkiewicz, Micha; Czarnecki, Krzysztof: Framework-Specific Modeling Languages with Round-Trip Engineering. In: Model Driven Engineering Languages and Systems, volume 4199 of LNCS. Springer Berlin Heidelberg, 2006.

[Ac15] Acceleo. `https://eclipse.org/acceleo/`, October 2015.

[Ah07] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D.: Compilers: Principles, Techniques, & Tools. Addison-Wesley series in computer science. Pearson Addison-Wesley, 2007.

[CC90] Chikofsky, E.J.; Cross, J.H., II: Reverse engineering and design recovery: a taxonomy. Software, IEEE, 7(1), 1990.

[Fo10] Fowler, Martin: Domain-Specific Languages. Addison-Wesley Signature Series. Pearson Education, 2010.

[Ga95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.

[Ha15] Haber, Arne; Look, Markus; Mir Seyed Nazari, Pedram; Navarro Perez, Antonio; Rumpe, Bernhard; Völkel, Steven; Wortmann, Andreas: Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development. SciTePress, 2015.

[HNR15] Hölldobler, Katrin; Nazari, Pedram Mir Seyed; Rumpe, Bernhard: Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In: 15th Workshop on Domain-Specific Modeling. http://dsmforum.org/events/DSM15/Papers/DSM15Proceedings.pdf, 2015.

[Je15] JetBrains MPS. https://www.jetbrains.com/mps/, October 2015.

[JMS08] Jörges, Sven; Margaria, Tiziana; Steffen, Bernhard: Genesys: service-oriented construction of property conform code generators. Innovations in Systems and Software Engineering, 4(4):361–384, 2008.

[KRV10] Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: MontiCore: A Framework for Compositional Development of Domain Specific Languages. International Journal on Software Tools for Technology Transfer, 12, 2010.

[Lo13] Look, Markus; Navarro Pérez, Antonio; Ringert, Jan Oliver; Rumpe, Bernhard; Wortmann, Andreas: Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In: Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC). volume 1102 of CEUR Workshop Proceedings, 2013.

[Me15] Metaborg Spoofax. http://metaborg.org/spoofax/, October 2015.

[MER99] Medvidovic, Nenad; Egyed, Alexander; Rosenblum, David S.: Round-Trip Software Engineering Using UML: From Architecture to Design and Back. In: Proceedings of the 2nd Workshop on Object-Oriented Reengineering. ACM Press, 1999.

[NRR15] Nazari, Pedram Mir Seyed; Roth, Alexander; Rumpe, Bernhard: Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table. In: 15th Workshop on Domain-Specific Modeling. http://dsmforum.org/events/DSM15/Papers/DSM15Proceedings.pdf, 2015.

[OM15] OMG UML Specification. http://www.omg.org/spec/UML/2.5/, October 2015.

[Pa10] Parr, Terence: Language Implementation Patterns: Create Your Own Domain-specific and General Programming Languages. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010.

[Ru11] Rumpe, Bernhard: Modellierung mit UML. Xpert.press. Springer Berlin, 2nd edition edition, September 2011.

[Ru12] Rumpe, Bernhard: Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring, volume 2nd Edition. Springer, 2012.

[Xt15] Xtext. https://eclipse.org/Xtext/, October 2015.

[ZR11] Zschaler, Steffen; Rashid, Awais: Towards Modular Code Generators Using Symmetric Language-aware Aspects. In: Proceedings of the 1st International Workshop on Free Composition. ACM, 2011.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 141

Modeling Variability in Template-based Code Generators for Product Line Engineering

Timo Greifenberg[1] , Klaus Müller[1], Alexander Roth[1], Bernhard Rumpe[1], Christoph Schulze[1], Andreas Wortmann[1]

Abstract: Generating software from abstract models is a prime activity in model-driven engineering. Adaptable and extendable code generators are important to address changing technologies as well as user needs. However, they are less established, as variability is often designed as configuration options of monolithic systems. Thus, code generation is often tied to a fixed set of features, hardly reusable in different contexts, and without means for configuration of variants. In this paper, we present an approach for developing product lines of template-based code generators. This approach applies concepts from feature-oriented programming to make variability explicit and manageable. Moreover, it relies on explicit variability regions (VR) in a code generator's templates, refinements of VRs, and the aggregation of templates and refinements into reusable layers. A concrete product is defined by selecting one or multiple layers. If necessary, additional layers required due to VR refinements are automatically selected.

Keywords: Model-Driven Engineering, Code Generator Development, Variability Modeling

1 Introduction

Engineering complex software systems introduces a conceptual gap between the problem domains and the solution domains of discourse [FR07]. Model-driven engineering (MDE) aims to bridge this gap by lifting abstract models to primary development artifacts. Deriving executable software from models requires extensive handcrafting or code generators. Thus, generating software from abstract models is a prime activity in MDE and many domains have adopted code generation [RR15].

Although reuse is of essence in software engineering, most code generators are monoliths developed for a very specific purpose (such as a certain target platform with specific features) that do not consider reuse or variability as their primary focus. Reusing such code generators in different contexts with different requirements or features is hardly feasible and thus impedes code generator development. One approach to handle variability in such monolithic code generators is to create code generator variants via informal reuse [Jö13] such as copy-paste. In this scenario, the original code generator variant is copied and all required changes are applied to the copy of the variant. The main downside of this approach is that generator changes might need to be applied to all generator copies. This is laborious and error-prone. An alternative to that is to use specific code generation frameworks with built-in support for handling variability [Ac15, Xt15]. Even though this alternative does not result in monolithic code generators, the resulting code generator variants are bound

[1] RWTH Aachen University, Software Engineering, Germany, http://www.se-rwth.de

to a specific code generation framework, which might not be feasible for legacy code generators. Additionally, the provided approaches rely on language specific approaches for implementing variability, e.g. design patterns.

The goal of this paper is to present an approach to develop code generator product lines (CGPLs), which is explicitly designed to handle variability in code generators and which can be applied to any code generator framework. To implement variability in code generators, the approach is based on explicit variability regions (VRs) and the aggregation of templates into reusable layers. Each VR can refine one or multiple VRs from a different layer. A concrete code generator variant is configured by selecting one or multiple layers. In addition to that, further layers are selected automatically, if this is required by the VR refinements. The resulting layers are composed to create a concrete variant. For defining and selecting layers, we provide two domain-specific languages (DSLs).

This idea is rooted in feature-oriented programming (FOP), an implementation technique from classical software product line (SPL) development [Ap13]. We extend the notion of FOP layers [SB98] over templates and define how (parts of) templates can be reused with these layers. The benefit of applying ideas of FOP to CGPL development is that the underlying concept is decoupled from specific template languages and can be applied to any code generator.

In the remainder, Section 2 introduces the variability concepts our approach relies on and Section 3 describes the product configuration mechanisms for code generators. Afterwards, Section 4 demonstrates the application of our approach to a code generation framework. Then, we compare our approach to the informal (copy-paste) approach for creating CGPLs in a case study in Section 5. Subsequently, related work is presented in Section 6 and, finally, Section 7 concludes this contribution.

2 Variability Concepts in Code Generator Product Lines

Code generator product lines and common SPLs are both founded on a set of components that are used to create a concrete code generator product or a software product [CN12, PBL05, RR15]. The main difference is that a code generator product is a SPL on its own, since it generates a variety of software products that are similar, and thus shares generator components potentially in different variants [BS99]. As in SPLs, a concrete code generator product, which is referred to as a variant, is a set of selected components with additional adaptations and customizations.

Feature-oriented programming (FOP) [Ap13] is an approach to implement SPLs that is based on building software systems by composing features. A feature represents a configurable unit of a software system that represents a requirement-satisfying design decision [ALS06, Ap13]. Each feature is arranged in a layer [SB02, BSR03, ALS05] that contains artifacts. In order to reuse existing functionality and to successively add new features by adapting existing artifacts, an artifact may refine multiple other artifacts [Ap13]. In FOP, a refinement adds new code to an existing artifact, e.g., adds a new variable to a Java class. Figure 1 shows an example for a stack of three layers with refinements.

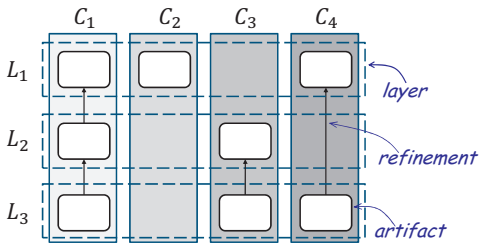


Fig. 1: Example for layers: three vertical layers L_1 to L_3 refine four artifacts C_1 to C_4.

In this example, the first layer contains three artifacts. The second layer contains a refinement for the artifact C_1 and a new artifact C_3. Finally, the third layer contains a refinement of the refinement for C_1, a refinement for C_3 and a refinement of C_4 from the first layer. By merging layers, different variants of a software system are formed. As the used layers contain code generator artifacts, the layers are subsequently called *code generator layers (CGLs)*. As shown in this example, FOP relies on artifact refinements. This is feasible for object-oriented languages but becomes challenging for template languages, as they may differ inherently. Thus, a concept for applying FOP to template languages is needed.

2.1 Variability Regions and Variability Region Refinements

Variability regions (VRs) provide a template language independent approach to apply concepts of FOP to code generators. A VR represents an explicitly designated region in an artifact that has to be uniquely addressable by an appropriate signature. We distinguish between three types of VRs. First, variability regions are explicitly marked in some way and contain content within an artifact. This, for instance, allows to group a designated part of a template as one VR, which can be refined. Second, variability regions are explicitly marked but are empty, i.e. do not contain any content. Such VRs can be used for future extensions. Third, the complete artifact is regarded as one VR.

For each VR, we define three different refinement operations. First, a `replace` operation completely replaces a VR with some other content. In this case, a certain VR is provided that substitutes the original VR. For example, template code for a Java method can be replaced with a new implementation. Second, content can be added before a VR and, third, content can be added after a VR. Semantically, `before` and `after` mean that specific content should be included before or after a VR. This shares many phenomena with aspect-oriented programming (AOP) [Ki97] applied on templates.

When dealing with `replace` operations, the effect of a `replace` operation to the content added before and after a VR has to be addressed. In this work, VRs are simply replaced but the before and after content, which may have been added, is kept. When the content that replaces a VR or that is added contains VRs, the new content with the VRs is regarded as a complete unit with all replacements and `before` and `after` operations. In consequence, all existing before and after contents have to be composed.

3 Code Generator Variant Configuration and Generation

A CGPL consists of a number of CGLs and each CGL contains a number of templates. Before a concrete product of a CGPL can be generated, it has to be defined which refinement operations are performed in which CGL, i.e., which VR contained in a template from a CGL refines which VR contained in a template from another CGL. Based on such a definition of the refinement operations, a concrete code generator variant of the CGPL can be configured and finally generated.

3.1 Layer Definition

In our approach, all files encapsulated in a concrete CGL of the CGPL are stored in a specific file system directory, comparable to the work in [ALS05]. The refinement operations that are performed for each CGL are modeled in one layer definition model. To define a layer definition model, we provide a simple domain-specific language (DSL) called Layer Definition Language (LDL). Using LDL, it can be defined which CGL refines which other CGL and which concrete refinement operations are performed. LDL allows for modeling the three refinement operations we introduced in Section 2.1:

- `A replaces B`: The VR with signature `B` is replaced by the VR with signature `A`.
- `A before B`: The VR with signature `A` is added before the VR with signature `B`.
- `A after B`: The VR with signature `A` is added after the VR with signature `B`.

An example for a layer definition model defined using LDL is shown in List. 1. At first, this example states that CGL `factoryVariant` refines CGL `baseVariant` (l. 1). Subsequently, the layer definition model defines which concrete refinement operations are performed (ll. 2 – 3). As explained in Section 2.1, we require each VR to be uniquely identifiable by its signature. In List. 1, the first refinement operation (l. 2), which is a `replace` operation, refers to the signatures `EntityExt:AdditionalMethods` and `ClassMain:Methods`. By default, each VR signature starts with the path to the artifact containing the VR (relative to the CGL directory) and its name. Hence, the first refinement operation expresses that the artifact `EntityExt` contains a VR `AdditionalMethods` and that this VR replaces the VR with name `Methods` contained in artifact `ClassMain`. Signatures for VRs can also be constructed in different ways, as long as it is possible to uniquely identify the artifact and the VR in that artifact. The second refinement operation (l. 3) states that the VR `ClassCopyright`, which represents a complete template, is added before the VR `Class`, which represents a complete template too. If the CGPL contains other CGLs with refinement operations, these have to be defined in the layer definition model too.

As this example already indicates, layer definition models are not restricted to particular types of languages. The only decision that has to be made is how to uniquely identify a VR within an artifact written in a particular language.

LayerDefinition

```
layer factoryVariant refines baseVariant {
  EntityExt:AdditionalMethods replaces ClassMain:Methods;
  ClassCopyright before Class;
}
```

List. 1: A layer `factoryVariant` that defines two refinements of the layer `baseVariant`.

3.2 Variant Configuration

Based on the layer definition model, a concrete generator variant can be configured by defining which CGLs of the CGPL should be selected. As a consequence of this, a concrete generator variant will be created which results from composing the VRs of the selected CGLs with the VRs they refine and all other not refined VRs of the selected layers. This procedure is repeated for the refined CGLs until a CGL is traversed which does not refine any other CGL. To configure a concrete generator variant, we define a product configuration model using a simple DSL called Product Configuration Language (PCL). In PCL, the name of the resulting concrete generator variant has to be defined and it has to be stated which CGLs should be selected. In each PCL, at least one CGL must be selected. Moreover, it can optionally be defined into which output folder the artifacts of the resulting generator variant are written.

An example for a product configuration model defined in PCL is shown in List. 2. According to this configuration, the resulting generator variant will be called `FactoryGenerator` and this variant is constructed by selecting the CGL `factoryVariant`. Moreover, the artifacts of `FactoryGenerator` would be written to the output folder `gen`. To infer which CGLs need to be composed to create the `FactoryGenerator`, the layer definition model needs to be analyzed. In this example, the layer definition model is given in List. 1 and it indicates that the CGL `factoryVariant` refines CGL `baseVariant`. Thus, both CGLs need to be composed to create `FactoryGenerator`. However, before a concrete generator variant can be composed, it has to be ensured that the layer definition model is valid, i.e., a set of layers can be computed and VR refinements are unambiguous. Validation is required to ensure that the selected code generator product can actually be build.

ProductCfg

```
generator FactoryGenerator {
  output = "gen";
  layers = "factoryVariant";
}
```

List. 2: Example for a product configuration model selecting layer `factoryVariant`

To validate the layer definition model, we map it to colored directed graphs, where each vertex represents a VR, each edge a refinement, and the color represents the layer a VR belongs to. First, the refinement operations for the selected layers are processed. For each refinement two vertices are introduced, if they are not already existing in the graph: one for the refining VR and one for the refined. The added vertices represent the VRs with all

their contained VRs. Additionally, a directed edge between the two vertices is created. It points from the refining VR to the refined. Each vertex that represents a VR of the current layer is colored in a particular color, that represents the layer. The other vertex is colored in another color that represents the other layer. After processing the refinement operations of the selected layers, the graph is traversed. Each time a new vertex with a color that has not yet been processed is found, the layer definition model is processed as described above.

A layer definition model does not induce any conflicts if and only if: 1) for any two vertices v_i^q and v_j^c with colors q and c and and a path (v_i^q, v_j^c) there exists no other path (v_j^c, v_i^q), i.e., the graph does not contain a cycle and 2) there exists no other vertex v_f^g with color g such that (v_f^g, v_j^c) holds, i.e., a VR is not refined by multiple VRs. A cycle in the graph makes it impossible to perform the composition automatically, as it results in an infinite loop of refinements with no dedicated end point. Furthermore, if multiple CGLs are selected and in this selection, a VR is refined by multiple VRs, it cannot be automatically decided which of these multiple refinements should actually take place in the composition. In both situations, it is necessary to resolve the problem manually. To derive a valid configuration, we can employ any graph traversal algorithm and select the different colors of the visited vertices, which represent the different layers.

In Figure 2, an example for a graphical representation of a layer definition model is given on the left-hand side. The resulting graph structure is shown on the right-hand side. It is assumed that artifacts of layer L_1 are colored in purple (p), artifacts of L_2 are colored in orange (o) and artifacts of L_3 are colored in blue (b). We further assume, that the layer L_3 is manually selected and, therefore, layers L_2 and L_1 are automatically selected because of their refinements. In this example, there is a path (C_1^b, C_1^p) as L_3 contains a refinement of a refinement of C_1 from L_1. In addition, as L_2 is automatically selected, the refinements (C_2^b, C_2^p) and (C_2^o, C_2^p) produce a conflict.

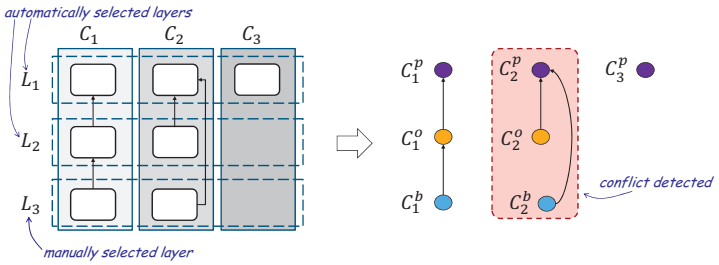


Fig. 2: A three-layered example, where layer L_3 is manually selected and layers L_2 and L_1 are automatically selected because of their refinements. This selection introduces a cycle and, thus, a conflict.

3.3 Building Variants by Composition

Based on the layer definition model and the product configuration model, a concrete code generator variant is created. To this effect, the templates, and corresponding refinements are composed. A template that is not refined in one of the relevant layers is used as is for the resulting code generator variant.

Refining templates requires proper application of refinement operations prior to variant composition. To compose a variant, we first need to (a) transitively determine all layers that have to be additionally selected because of the refinements and (b) compose the resulting layers to define the code generator variant.

In general, there are two options on composing VRs. The first option is to perform composition at run-time of the generator, called *generation-time*. In this case the VR operations are executed at generation-time. This means that no VRs are created but support for generation-time execution of VR operations is required. As an alternative, the VRs can be composed by creating new VRs which contain the composition results. The main question that needs to be answered when applying this latter approach is how to deal with `before` and `after` operations. To avoid that the generator framework has to be extended to be able to handle these operations at generation-time, on template level, `before` and `after` operations can be replaced by template inclusion statements in the according template language.

The composition of two layers means that all contained artifacts and their VR refinements are composed. Two layers are composed if and only if there is at least one VR refining a VR in the opposite layer, according to the understanding of composition as defined for FOP [Ap13]. If more than two layers are involved in the composition, then we process all refinements for one VR sequentially in a bottom-up way. If in this sequence a refinement is a `replace` operation, then the VR being replaced is substituted by the VR replacing it. Moreover, if a refinement in a sequence denotes a `before` or `after` operation, then the refining VR is added before (respectively after) the refined VR.

An algorithm for performing this composition would start visiting all selected layers and then the automatically added layers. In each layer, every refinement is considered in a bottom-up way, i.e., only the outgoing refinements refining a VR are considered.

4 Demonstrating Example for Variability Regions

In this section, we demonstrate the application of our approach to the code generator framework openArchitureWare using Xpand [Xp15] as a template language. Motivated by an industrial use case (see Section 5), the openArchitectureWare framework in version 3.0.1 has been chosen.

4.1 Example Description

In this example, we consider a code generator that processes a class diagram (CD) as input and translates every class into a Java class with the same name. Each attribute of a class is translated to a Java variable with a mutator and an accessor method. It also adds a public constructor with an argument list containing all attributes defined in the class. For demonstration purposes, we assume the input CD contains the class `Person` with the attribute `name` of type `String`. On the left-hand side, Figure 3 shows a CD of the resulting generated class.

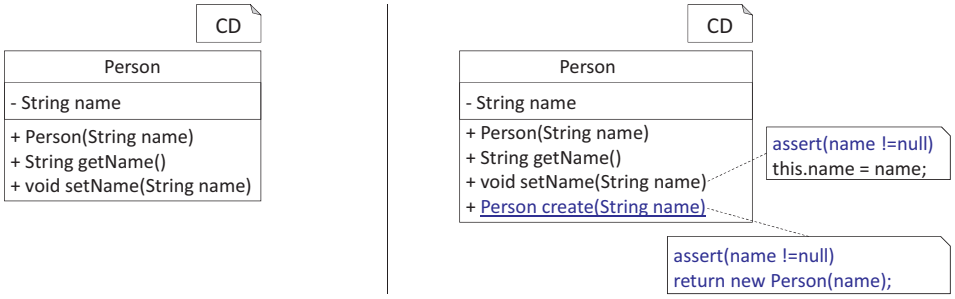


Fig. 3: Overview of the originally generated output (left) and the required output (right).

Another context requires to generate the code for classes differently. Instead of writing a new code generator from scratch or copying the original one, a new variant of the existing generator should be created. This variant should validate the argument passed to the mutator methods and produce a factory method that asserts proper creation arguments. A CD of the resulting output is depicted in Figure 3 on the right-hand side. Here, the generated class name corresponds to the input model's class name, it features an assertion in `setName()`, and provides a factory method `create()` for `Person` objects that asserts that the value passed for name actually exists. Please note that, usually the constructor visibility would be changed too, to prevent others from invoking the public constructor directly. However, due to limitations of space, we omitted this part and assumed that the constructor visibility is not changed. To achieve this kind of extensibility, our approach lifts the code generator to a CGPL by explicitly managing variability.

4.2 openArchitectureWare

The Xpand template language allows to split templates into multiple blocks. Such blocks begin with the keyword `DEFINE` and, thus, we henceforth refer to these as `DEFINE` blocks. Each `DEFINE` block is identified by a name and is defined for a specific type of input model element, called *meta model class*. For instance, all concrete classes of the CD in our example are represented by the meta model class `MMClass`.

List. 3 shows an excerpt of the realization of our example in Xpand. For the sake of brevity, only those parts are shown that are relevant for the refinement of the template. The first `DEFINE` block with name `ClassImpl` is defined for the meta model class `MMClass`. If this `DEFINE` block is invoked for a concrete class, a new Java file is created for that class, indicated by the `FILE` statement (l. 2). Expressions encapsulated in [...] lead to the invocation of the according methods of the meta model classes. The results of these invocations are inserted into the output at the current location.

To generate the implementation for a class, the `DEFINE` blocks `Constructor` (l. 4) and `FurtherMethods` (l. 6) are invoked for the class and `SetterMethod` (l. 5) is invoked for each attribute of the class by using the `EXPAND` statement. The string that is constructed in the according `DEFINE` blocks is inserted into the output at the current location.

Xpand

```
1 [DEFINE ClassImpl FOR MMClass]
2   [FILE ... Name".java"]
3   class [Name] {
4     [EXPAND Constructor]
5     [EXPAND SetterMethod FOREACH Attribute]
6     [EXPAND FurtherMethods]
7   }
8   [ENDFILE]
9 [ENDDEFINE]
10
11 [DEFINE Constructor FOR MMClass]
12   public [Name](...) {
13     [EXPAND ConstructorImpl]
14   }
15 [ENDDEFINE]
16
17 [DEFINE SetterMethod FOR MMAttribute]
18   public void set[UpperCaseName]([Type] [Name]) {
19     [REM]BEGIN VR:SetterMethodBody[ENDREM]
20       this.[Name] = [Name];
21     [REM]END VR:SetterMethodBody[ENDREM]
22   }
23 [ENDDEFINE]
24
25 [DEFINE FurtherMethods FOR MMClass]
26 [ENDDEFINE]
```

List. 3: Template `Class` (in Folder `base`) showing an excerpt of the base template for the translation of CDs into Java code realized with Xpand.

4.3 Mapping Variability Regions to Templates

In Section 2, we introduced three kinds of VRs: non-empty VRs that refer to a particular region within an artifact, empty VRs for future extensions and the VR representing the complete artifact. In Xpand, non-empty VRs can be introduced by defining non-empty `DEFINE` blocks and, accordingly, an empty VR can be introduced by declaring an empty `DEFINE` block. The most important aspect of every VR is that it has to be uniquely identifiable through its signature. The signature of a `DEFINE` block can be derived by the path to the template and its name. If multiple `DEFINE` blocks with the same name exist in one template, the meta model class of such a `DEFINE` block has to be stated in the signature as well. Otherwise, it cannot be differentiated between the different blocks with the same name. In addition, the complete template represents a VR as well with the path to the template and its name representing the signature of this VR.

Besides interpreting every `DEFINE` block as one VR, it is possible to introduce further VRs into Xpand templates explicitly by using, e.g., named comments around the corresponding region in the template. The advantage of using comments for this is that the template

language does not have to be extended and this approach is applicable to all template languages supporting comments. This approach is comparable to utilizing protected regions for integrating handwritten and generated code [Gr15], as there, comments mark the regions into which handwritten code can be inserted. List. 3 shows an example in lines 19 to 21, in which the body of the setter method is contained in the VR `SetterMethodBody`. The start and the end of the VR `SetterMethodBody` is denoted through comments, represented by `REM` and `ENDREM`, in which the name of the VR is defined. This comment-based approach is used here only for demonstration purposes, to illustrate how it can be applied. When using Xpand, instead, a separate `DEFINE` block could have been used as well. Even though this approach allows for introducing any kind of VR into a template, this approach is rather fragile, as a comment can be changed by accident easily. Moreover, a template might contain several other comments which makes it more difficult to identify VRs marked by comments.

4.4 Variability Region Refinements

Using the template introduced in List. 3, we show how VR refinement operations can be mapped to concepts in Xpand. This is done by using the layer definition model shown in List. 4. Moreover, List. 5 illustrates the refining template used for the example.

LayerDefinition

```
1 layer factoryVariant refines baseVariant {
2   base.ClassWithFact:FurtherMethods
3     replaces base.Class:FurtherMethods;
4
5   base.ClassWithFact:Method.SetterMethodBody
6     replaces base.Class:SetterMethod.SetterMethodBody;
7 }
```

List. 4: Layer definition model for Xpand realization.

As indicated by the first refinement operation, the VR `FurtherMethods`, contained in template `ClassWithFact` (ll. 1-8 of List. 5) which is located in folder `base`, replaces the empty VR `FurtherMethods` from template `Class` (ll. 25-26 of List. 3), which is located in folder `base` too. By means of this, the factory method `create()` is generated additionally.

Furthermore, the VR `SetterMethodBody` contained in the `DEFINE` block `Method` in template `ClassWithFact` (ll. 11-14 of List. 5) replaces the VR `SetterMethodBody`, contained in the `DEFINE` block `SetterMethod` in template `Class` (ll. 19-21 of List. 3). The comments denoting the start and the end of the VR `SetterMethodBody` are defined within a `DEFINE` block, as otherwise the resulting template would be syntactically wrong. This last refinement operation is responsible for introducing assert statements at the beginning of the setter methods. For this purpose, it takes advantage of the `INCLUDE-SUPER` statement, which we introduced to include the original content of the refined `DEFINE` block.

Consequently, the original content is inserted after the assert statement.

Xpand

```
[DEFINE FurtherMethods FOR MMClass]
  public static [Name] create(...) {
    [FOREACH Attribute AS at]
      assert([at.Name] != null);
    [ENDFOREACH]
    return new [Name](...);
  }
[ENDDEFINE]

[DEFINE Method FOR MMClass]
  [REM]BEGIN VR:SetterMethodBody[ENDREM]
    assert([Name] != null);
    [REM][INCLUDE-SUPER][ENDREM]
  [REM]END VR:SetterMethodBody[ENDREM]
[ENDDEFINE]
```

List. 5: Template `ClassWithFact` (in Folder `base`) showing an excerpt of the refining template for the translation of CDs into Java code realized with Xpand.

Please note that, it would not be possible to implement this variability using the XPand language constructs of the used XPand version - only later versions of XPand provide means to customize a code generator. Hence, without our approach, a copy of the original code generator variant would have to be created to develop the shown code generator variant. The decision to use this particular XPand version was rooted in the fact that this version was used in a real-world code generator to which we applied our approach in a case study (see Section 5).

5 Industrial Case Study

The approach has been applied to a large real-world Java code generator which processes UML CDs as input. For the contained classes, it generates, among other things, Java classes with mutator and accessor methods. Moreover, each Java class contains additional inner classes and accessor methods that expose the data in a different way and allow a special access to the Java fields. This code generator variant is in the following referred to by *OV1*. Besides this existing code generator *OV1*, a variant of this code generator *OV2* should be build which:

- does not generate the additional inner classes and special access methods.
- does not generate a normal Java field for all UML associations of the corresponding UML class but which generates a field of a special type for UML associations to UML classes tagged with a specific stereotype.
- names the resulting classes according to the originally named classes but with a new suffix, to be able to differentiate between the original and the new classes easily.

The objective of our case study is to demonstrate the usefulness and applicability of our approach to implement a CGPL for a real-world code generator and to compare it to the classical informal approach (copy-paste) for creating CGPLs. For this purpose, we derived the following research questions:

- Is it feasible to apply the approach to establish a CGPL for real-world code generator variants?
- Is the application of the approach superior to the informal reuse of code generators through copy-paste in terms of complexity of the involved artifacts?

5.1 Applicability to Real-World Code Generators

In order to better understand the usefulness of the approach, we first implemented the variant *OV*2 through informal reuse by doing copy-paste of *OV*1. Then, we applied our approach to realize both generator variants with our approach. For this purpose, we defined a CGL *NV1* which contains the common parts of *OV1* and *OV2*. Moreover, we defined a CGL *NV2* which refines *NV1* in such a way that the generator resulting from the composition of *NV1* and *NV2* generates the same code as *OV1*. Analogously, we defined a CGL *NV3* which refines *NV1* such that the generator resulting from the composition of *NV1* and *NV3* generates the same code as *OV2*. Hence, we assumed that the code generator variants resulting from the composition with the base layer *NV1* must generate the same code as the original code generators - neglecting whitespaces for the sake of simplification.

Using our approach, we were able to derive two code generator variants which generate the same code as code generator variants which did not use our approach. In particular, the presented refinement operations were sufficient to realize the CGPL. For these refinement operations, only `replace` refinement operations have been used, as the developers of the original code generator preferred these over introducing `before` or `after` operations.

5.2 Improvements over Informal Reuse

To answer the second research question, we compared the variants *OV1* and *OV2* with the variants *NV1*, *NV2* and *NV3*. To increase comparability, we removed those templates from *OV2* which were copied from *OV1* but not needed for that variant. However, we applied our concept not only on templates, but also on helper classes which can contain more complex functionality which can be accessed from templates. In this use case, helper classes were implemented in Java. The only refinement operation we used in this context was the `replace` operation, which expresses that the implementation of one helper method is replaced by the implementation of another helper method.

To perform the comparison, we measured the templates lines of code (TLOC) and the helper lines of code (HLOC) for *OV1*, *OV2*, *NV1*, *NV2* and *NV3*. To compare our approach with the copy-paste approach, we compared the total TLOC and HLOC of *OV1* and *OV2* with that of *NV1*, *NV2* and *NV3*.

	OV1	OV2	Σ_O	NV1	NV2	NV3	Σ_N
TLOC	5563	2260	7823	1882	4000	349	6231
Number `DEFINE`	327	146	473	189	267	37	493
Number refined `DEFINE`	-	-	-	-	94	36	130
HLOC	929	929 (665)	1858 (1594)	630	330	49	1009
Number helper	100	100 (79)	200 (179)	78	34	5	117
Number refined helper	-	-	-	-	8	8	16

Tab. 1: Case study results: TLOC and HLOC for the different variants

Table 1 gives an overview over the measured values for the original generator *OV1* and the variant *OV2* created through copy-paste of *OV1* and the generator variants obtained by using our approach. The primary numbers relevant for this comparison are TLOC and HLOC. For *OV2* two HLOC numbers are given: the first results from simple copy-paste of the original helpers, the second number refers to the case that only the helpers used by the variant are counted. Thus, the existing helpers have been analyzed and the helpers not needed were removed. For *NV2* and *NV3*, the number of helpers refers to the number of additionally introduced helper methods. Σ_O refers to the sum of the values of both variants *OV1* and *OV2*. Accordingly, Σ_N refers to the sum of the values for the variants *NV1*, *NV2* and *NV3*.

As can be seen in Table 1, we can reduce the TLOC size to approximately 79% of the original code generators using our approach. Moreover, we can reduce the HLOC size to approximately 54% respectively 63% of the original code generators.

In addition to that, Table 1 shows that the total number of `DEFINE` blocks is comparable for both variants. Even though `DEFINE` blocks can potentially be reused by multiple variants, this effect does not become apparent in this case, as only two generator variants are created and for each refinement of a `DEFINE` block, one `DEFINE` was introduced, increasing the total number of `DEFINE` blocks. For helper methods, a significant reduction can be observed, as most helper methods can be reused by both generator variants and only few refinements were necessary.

6 Related Work

Different annotative, compositional, and transformational modeling approaches have been proposed to express variability in the solution space [Sc12]. Annotative approaches specify all variants in one model. Compositional approaches combine different model fragments to derive a specific variant [HW07, NK08]. Delta modeling [Ha11] applies transformations to a core model. Only few of them have been successfully applied to CGPLs.

In the following, we present existing approaches to address variability in code generators with a special focus on existing code generator frameworks and how they support variability. The concepts we presented are independent of a concrete code generator framework

and template language. Another difference to most existing approaches is that arbitrary regions can be marked as VRs.

In the Genesys [JMS08] framework, new generators are established by composing existing Service Independent Building Blocks (SIBs), the atomic unit provided for composition. This approach has been evaluated in many case studies: in most cases, new generators could be derived by the introduction of a small set of new SIBs and a slightly modified composition. This specific mapping represents one point of variation, which can easily be adapted for different targets. The main part of variation are the SIBs, which can be modified via configuration parameters, via a modification of the their execution flow or by replacing a service adapter, which contains execution code for a specific task. In contrast to our approach, Genesys defines a set of different explicit concepts (parameter, service adapters, outgoing branches) to achieve the necessary variation. Our proposed approach of VRs allows to introduce variation points on different kinds of development artifacts and the related `before` and `after` operations can be used to manipulate the execution flow where necessary, too. This way it is also possible to apply variation points to templates, while in [JMS08] templates are modified directly and no variation points are introduced on that level.

[VG07b] highlights the necessity to combine model-to-model transformations and template-based code generation to perform efficient code generation. They suggest that all structural differences on model level should be handled by the transformation layer. [PT02] follow this by pointing out that the generator should handle only two kinds of variation: target variation and the establishment of higher-level primitives based on low-level primitives. Our approach does not provide a guideline on which level which kind of variation should be established, but represents a general concept, to be able to apply variation points where required. If a model-to-model transformation is performed via Java helper classes, corresponding variation can also be applied on that level.

Acceleo [Ac15] provides the concept of dynamic overriding to customize existing generators. To dynamically override templates, a module (which can comprise multiple templates) must extend a module of the existing generator. The extending modules are treated with a higher priority than overridden modules. Thus, the overriding template is invoked instead of the existing template. Templates can only be exchanged as a whole, no variation points can be introduced inside a template.

The template language Xpand supports the customization of code generators using aspect-oriented programming (AOP) [VG07a]. Aspects can be provided which contain template code that is, e.g., invoked instead of code contained in a specific block in the template. Although the original template definition is intercepted, the original overridden template code can still be called in the aspect code [El11]. Our approach is motivated by the concepts applied in Xpand. The main difference to our approach is that our approach does not require support for AOP in the code generator. In Xpand's successor Xtend [Xt15], code generators are composed of extension methods. To customize a code generator written in Xtend, any extension method of a code generator can be exchanged by means of dependency injection. However, these concepts are completely based on language constructs. In contrast, our approach is more general and can be realized with different languages.

7 Conclusion

Monolithic code generators are hard to adapt to new requirements and target platforms and, thus, are hardly reusable in different contexts, as they are not designed for adaptations. To overcome this limitation regarding customizations, code generator variability needs to be handled as a primary concern.

We have presented an approach for modeling variability in template-based code generators. This approach relies on *variability regions (VR)* that define extension points in artifacts. Furthermore, since it is an extension of feature-oriented programming, the artifacts are structured in layers that represent code generator features. We additionally define three refinement operations to extend VRs. In order to extend a code generator with a new feature, a new layer can be introduced and existing VRs can be refined. The benefit of the proposed concept is that it is independent of any language that is used for code generator development. We achieve this by introducing a layer definition model language that can be used with any other language. By means of this, the approach facilitates reusing and customizing code generators.

References

[Ac15] Acceleo website. `http://www.eclipse.org/acceleo/`, October 2015.

[ALS05] Apel, Sven; Leich, Thomas; Saake, Gunter: Aspect Refinement and Bounding Quantification in Incremental Designs. In: APSEC. IEEE, 2005.

[ALS06] Apel, Sven; Leich, Thomas; Saake, Gunter: Aspectual Mixin Layers: Aspects and Features in Concert. In: ICSE. ACM, 2006.

[Ap13] Apel, Sven; Batory, Don; Kästner, Christian; Saake, Gunter: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer-Verlag, October 2013.

[BS99] Batory, Don; Smaragdakis, Yannis: Building Product-Lines with Mixin-Layers. In: ECOOP Workshops. Springer, 1999.

[BSR03] Batory, Don; Sarvela, Jacob Neal; Rauschmayer, Axel: Scaling Step-wise Refinement. In: ICSE. IEEE, 2003.

[CN12] Clements, Paul; Northrop, Linda: Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., February 2012.

[El11] Elsner, Christoph; Groher, Iris; Fiege, Ludger; Völter, Markus: Model-Driven Engineering Support For Product Line Engineering. In: Aspect-Oriented, Model-Driven Software Product Lines - The AMPLE Way. Cambridge University Press, 2011.

[FR07] France, Robert; Rumpe, Bernhard: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering 2007 at ICSE. IEEE, 2007.

[Gr15] Greifenberg, Timo; Hoelldobler, Katrin; Kolassa, Carsten; Look, Markus; Mir Seyed Nazari, Pedram; Mueller, Klaus; Navarro Perez, Antonio; Plotnikov, Dimitri; Reiss, Dirk; Roth, Alexander; Rumpe, Bernhard; Schindler, Martin; Wortmann, Andreas: A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In: MODELSWARD. Scitepress, 2015.

[Ha11] Haber, Arne; Kutz, Thomas; Rendel, Holger; Rumpe, Bernhard; Schaefer, Ina: Delta-oriented Architectural Variability Using MontiCore. In: ECSA. ACM, 2011.

[HW07] Heidenreich, Florian; Wende, Christian: Bridging the gap between features and models. In: AOPLE. 2007.

[JMS08] Jörges, Sven; Margaria, Tiziana; Steffen, Bernhard: Genesys: service-oriented construction of property conform code generators. Innovations in Systems and Software Engineering, 4(4), 2008.

[Jö13] Jörges, Sven: Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach, volume 7747 of LNCS. Springer, 2013.

[Ki97] Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Lopes, Cristina; marc Loingtier, Jean; Irwin, John: Aspect-Oriented Programming. In: ECOOP. volume 1241 of LNCS. Springer, 1997.

[NK08] Noda, Natsuko; Kishi, Tomoji: Aspect-Oriented Modeling for Variability Management. In: SPLC. IEEE, 2008.

[PBL05] Pohl, Klaus; Böckle, Günter; Linden, Frank van der: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, 2005.

[PT02] Pohjonen, Risto; Tolvanen, Juha-Pekka: Automated production of family members: Lessons learned. In: PLEES. Fraunhofer IESE Technical Report IESE-Report 056.02/E, 2002.

[RR15] Roth, Alexander; Rumpe, Bernhard: Towards Product Lining Model-Driven Development Code Generators. In: MODELSWARD. SciTePress, 2015.

[SB98] Smaragdakis, Yannis; Batory, Don S.: Implementing Layered Designs with Mixin Layers. In: ECOOP. volume 1445 of LNCS. Springer-Verlag, 1998.

[SB02] Smaragdakis, Yannis; Batory, Don: Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs. ACM Transactions on Software Engineering and Methodology, 11(2), April 2002.

[Sc12] Schaefer, Ina; Rabiser, Rick; Clarke, Dave; Bettini, Lorenzo; Benavides, David; Botterweck, Goetz; Pathak, Animesh; Trujillo, Salvador; Villela, Karina: Software diversity: state of the art and perspectives. STTT, 2012.

[VG07a] Völter, Markus; Groher, Iris: Handling Variability in Model Transformations and Generators. In: DSM. `http://dsmforum.org/events/DSM07/papers/voelter.pdf`, 2007.

[VG07b] Völter, Markus; Groher, Iris: Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development. In: SPLC. IEEE, 2007.

[Xp15] Xpand website. `https://eclipse.org/modeling/m2t/?project=xpand`, October 2015.

[Xt15] Xtend website. `http://www.eclipse.org/xtend/`, October 2015.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 157

A Software Product Line of Feature Modeling Notations and Cross-Tree Constraint Languages

Christoph Seidl[1], Tim Winkelmann[1], Ina Schaefer[1]

Abstract: A Software Product Line (SPL) encompasses a set of closely related software systems in terms of common and variable functionality. On a conceptual level, the entirety of all valid configurations may be captured in a variability model such as a feature model with additional cross-tree constraints. Even though variability models are essential for specifying configuration knowledge, various notations for feature models and cross-tree constraints exist, which increases implementation effort when having to realize new tools for a different language. In this paper, we provide remedy to this problem by introducing an SPL to generate different variants of feature modeling notations and cross-tree constraint languages. We base our approach on the state of the art in various works and surveys on feature modeling to create a family of feature modeling notations with similar expressiveness as the original approaches. For our findings, we provide both conceptual configuration knowledge as well as a generative model-based realization. We further demonstrate the feasibility of our approach by generating feature modeling notations similar to those of various publications.

1 Introduction

A Software Product Line (SPL) encompasses a set of closely related software systems in terms of common and variable functionality. On a conceptual level, configuration knowledge may be captured in a *variability model* to describe configuration rules for all valid products. The most commonly used type of variability models are *feature models* [Ka90], which arrange *features* as configurable units along a tree-structured decomposition hierarchy. Commonly, language constructs are used to represent *optional* and *mandatory* features as well as *alternative-groups* allowing selection of exactly one feature and *or-groups* permitting selection of one or more features. Furthermore, cross-tree constraints [Ba05] may be used to further restrain configuration options.

Ever since the original introduction of feature models [Ka90], a great number of extensions has been made to the original notation to address various needs, such as attributes for features with finite and infinite domains [Cz02, CHE05, KOD10], cardinalities for groups and individual features [Ri02, CHE05, ME08, SSA14a] or configurable feature versions [SSA14a, SSA14c]. Likewise, languages for cross-tree constraints may be represented by different means using various subsets of propositional logic [Ba05] or the OCL [CE00, Cz02] (Object Constraint Language) as well as textual formulations [Ba05, KOD10] and graphical representation as additional edges in the feature model [HSVM00, SLW12]. Through these extensions for feature models and cross-tree constraints, a wide variety of different concerns of configuration problems can be addressed.

[1] Software Engineering Institute, Technische Universität Braunschweig, E-Mail: {c.seidl,t.winkelmann,i.schaefer}@tu-bs.de

However, the cluttering of different languages also entails problems in practice as, e.g., implementation effort is increased when having to realize new tools for a different language, which results in hampered progress and, presumably, less robust software.

Despite the differences in the various languages for feature models and cross-tree constraints, the languages encompass a significant level of commonality. In this paper, we exploit this fact to make a first step towards solving the aforementioned problems by devising an SPL to generate feature modeling notations and cross-tree constraint languages with a selected set of language constructs. We bootstrap feature modeling to conceptually capture the configuration knowledge of the software family. We further use SPL techniques to allow configuration of various different concrete languages for feature models and cross-tree constraints. We supply both theoretical background as to the capabilities of the different language constructs as well as technical realizations of the different concrete languages through generative model-based development.

With these contributions, it is possible to derive variants of feature modeling notations and cross-tree constraint languages that are similar to those of existing approaches and to treat them with procedures common in SPL engineering (e.g., family-based analyses). Furthermore, new combinations of feature modeling language constructs may yield previously non-existing notations to address the individual characteristics of specific use cases. In the future, this approach may be used to support data exchange between different notations by transforming one notation of a source system to an (at least) equally expressive notation with different characteristics of a target system.

The rest of this paper is structured as follows: Section 2 provides the criteria we used to select feature modeling notations we consider for our SPL. Section 3 analyzes 23 individual approaches from the state of the art in feature modeling (including constraint languages) as well as 4 surveys on feature modeling notations and categorizes the respective approaches by a number of distinctive characteristics. Section 4 presents our family of feature models and constraint languages that subsumes all of the analyzed approaches in expressiveness and even allows creation of previously non-existent notations. Section 5 demonstrates feasibility of our work by first applying the implementation of the presented feature modeling family to generate multiple variants with different expressiveness and then recreating the available examples of the inspected approaches. Finally, Section 6 closes with a conclusion and an outlook to future work.

2 Considered Work

A large number of similar yet different feature modeling notations exists which address a wide variety of different concerns. Developers have to find the right tool with the right notation for their respective SPL project. To help developers in making a conscious decision, an overview of existing notations and their dependencies is needed. As the different characteristics of feature modeling notations can themselves be represented as features and the dependencies can be expressed using constraints, we create an SPL that covers the variability of the inspected modeling notations. For this purpose, we define a family of feature modeling notations including cross-tree constraint languages to express dependencies

between the elements of a feature model. However, we do not claim completeness with regard to expressiveness or other properties such as succinctness or naturalness [SHT06] for *all* feature modeling notations. Nevertheless, we do cover a wide range of approaches used in practice [Ka90, Ka98, GFA98, HSVM00, vGBS01, Ri02, EBB05, CE00, Cz02, CHE04, BTRC05] as well as specific special-purpose extensions [ME08, SLW12, KSS13, SSA14a].

To select suitable feature modeling notations to analyze as basis for the presented software family, we applied the following criteria for selection: First, we included those feature modeling notations that have a particular high impact on further development. For this purpose, we considered all approaches included in Kang's keynote presentation from VaMoS'10 on 20 years of feature modeling [Ka10] as they are fundamental for many further feature modeling notations [Ka90, Ka98, GFA98, HSVM00, vGBS01, Ri02, EBB05, CE00, Cz02, CHE04, BTRC05]. We further included surveys that analyze state of the art in feature modeling [Jé12, BSRC10, SHT06, CHE04] to determine relevant feature modeling approaches. Second, we included approaches that are representative for various special-purpose extensions [ME08, SLW12, KSS13, SSA14a] in feature modeling. Third, we included textual variability languages, such as TVL [CBH11], Familiar [Ac13] and Clafer [BCW11], which can be represented with a meta-model similar to feature models. For work from the last 10 years, we mainly considered those extensions that explicitly provide a meta-model or grammar for their language and that introduce new concepts to feature models.

Existing surveys on feature modeling notations have presented contributions that are closely related to our work: Schobbens et al. [SHT06] also analyzed a great number of the high impact feature modeling notations in a formal way. During their analyses, they devised a language called *Varied Feature Diagrams (VFD)* [Sc07]. For VFD, they used expressive and succinct elements of feature modeling notations to build a combined modeling notation. That notation is as expressive and succinct as the other analyzed notations. Their approach is different from ours as we generate variants of feature modeling notations with capabilities tailored to the respective intended use instead of creating one monolithic notation for all use cases. Dhungana et al. [Dh13] allow the use of various different variability modeling notations and transform them into a uniform notion of configuration options. Hence, their approach is practical for the configuration process of different employed variability modeling notations where ours aims towards the modeling process. Hubaux et al. [HTH13] analyzed feature diagram languages with regard to separation and composition of concerns. They identified important concerns and purposes of feature diagram languages, how they are separated and composed in existing SPL approaches and other characteristics. Different notations we discuss in this paper have a direct impact on separation and composition of SPLs. Jézéquel [Jé12] surveys several modeling notations and explores the combination of these notations with artifacts in the product generation process. Lichter et al. [Li03] compare a number of feature modeling notations with a focus on the process and required input to make a particular feature modeling notation useful. While they acknowledge large commonalities but also differences of notations, they do not build an SPL to create different variants of feature modeling notations. Eichelberger and Schmid [ES13, ES14] present an analysis of textual variability languages. They provide an overview of the commonalities and differences of the notations

in order to provide information on the evolution of textual variability modeling languages and identifying common weaknesses for future research.

We improve over the mentioned literature reviews [Jé12, Li03, BSRC10, Ka10, Sc07] by not just listing and discussing different characteristics of feature modeling notations but by further presenting an SPL and a model-based realization to generate individual variants. The idea of our work is that the provided SPL can be used to create variants for representing configuration knowledge in a notation tailored to the concrete use case. However, at present, proofs for semantic equivalence of arbitrary variants are outside the scope of the paper. In the future, appropriate tools should be generated, such as an editor that supports the respective notations. As basis for this SPL, we selected 23 different feature modeling notations we analyze for distinct characteristics in the next section.

3 Analysis

Using the criteria presented in Section 2, we determined 23 different approaches for feature modeling notations and constraint languages to analyze regarding their common and distinctive characteristics as basis for a software family. Table 1 summarizes our findings and the following sections elaborate on characteristics of the analyzed approaches regarding notational concepts of the employed feature modeling and constraint languages.

3.1 Feature Modeling Notations

In the upper part of Table 1, we provide information on different language constructs provided by the examined approaches, which are explained in the following.

Mandatory Features represent commonalities that have to be included in a configuration if their parent feature is selected. All examined approaches support this language construct.

Optional Features represent variabilities that may or may not be included in a configuration. All examined approaches support this language construct.

Feature Cardinality specifies a minimum and maximum number for how often a feature may be selected as [m..n]. When using 1 as maximum cardinality [Cz02, CHE04, Ri02, ME08, SLW12, SSA14a], feature cardinality may be perceived as alternative to the explicit variation type for mandatory features (i.e., [1..1]) and optional features (i.e., [0..1]). Furthermore, Czarnecki et al. [Cz02, CHE04] use feature cardinalities to be able to represent multiple instances (e.g., [2..5]) of one and the same feature as *cloned features* by allowing maximum cardinalities greater than 1.

Attributes are named variables of features [Ri02, Cz02, CHE04, BTRC05] that refine configuration options so that, besides selection of features, concrete values for attributes may be chosen. Czarnecki et al. [Cz02, CHE04] and Benavides et al. [BTRC05] assign a specific *type* to the attributes, which specifies permissible values. In the literature, types of attributes are typically categorized into *discrete* (*finite* or *infinite*) and *continuous* domains.

Feature Versions include variability in time in feature models [SSA14a, ME08]. Mitschke et al. [ME08] support two versions per feature representing the state of the feature model's structure and its associated implementation but do not allow using them as configurable units. Seidl et al. [SSA14a, SSA14c] allow specification of multiple feature versions with interdependencies to make feature versions a configurable unit. Configurable versions may not adequately be represented using attributes as their relation cannot be specified properly.

Layers of feature models provide a separation of concerns for different sources of variability. Kang et al. [Ka98] use layers for *Capability*, *Operating Environment*, *Domain Technology* and *Implementation Technique*. Each layer may contain a set of separate feature models with relations to feature models of other layers. This increases the reuse of feature models and supports scalability.

External Features allow referencing of features that are defined in other feature models [vGBS01]. For example, this may be used in combination with layers of feature models when referencing features of other feature models [BCW11, Ab10, CBH11, Ro11, Ac13].

Binding Times specify at which time a feature may be or has to be configured. Typical binding times are at *compile time* or *run time* [GFA98, vGBS01, B3]. Griss et al. [GFA98] use attributes in the features to describe the binding time. Van Gurp et al. [vGBS01] use a label on the connector between features to distinguish the binding time.

Resource Mapping allows association of various resources with the features in a feature model [SLW12, KSS13, Th11]. Schroeter et al. [SLW12] provide a mapping of features to views, which show only selective parts of a feature model for different stakeholders of the feature model. Kowal et al. [KSS13] map priorities for the configuration and specific hardware to the features.

Alternative-Groups allow selection of exactly one of the contained features, which makes them mutually exclusive. All examined approaches support this language construct.

Or-Groups allow selection of at least one of the contained features. With the exception of Kang et al. [Ka90], all examined approaches support this language construct.

Group Cardinality specifies the minimum and maximum number of selectable features in that group as [m..n]. Hence, it may be perceived as an alternative to the explicit variation type of groups as alternative-groups (i.e., [1..1]) and or-groups (i.e., [0..n] for groups with n members) [Ri02, CHE04, SLW12, SSA14a]. In contrast to the explicit variation types, group cardinality supports further restrictions on selections in a group (e.g., [2..5]).

Multiple Groups describe the possibility that a feature can have more than one child group, e.g., a feature that has two alternative-groups. Many notations do not explicitly state whether they support multiple groups or not. Czarnecki and Eisenecker [CE00] appear to be the first who explicitly support multiple groups.

		Kang et al. 1990 [KCH+90]	Kang et al. 1998 [KKL+98]	Griss et al. 1998 [GFA98]	Hein et al. 2000 [HSVM00]	Czarnecki et al. 2000 [CE00]	Van Gurp et al. 2001 [vGBS01]	Riebisch et al. 2002 [RBSP02]	Czarnecki et al. 2002 [CBUE02]	Czarnecki et al. 2004 [CHE04]	Batory 2005 [Bat05]	Eriksson et al. 2005 [EBB05]	Benavides et al. 2005 [BTRC05]	Schobbens et al. 2007 [SHTB07]	Mitschke et al. 2008 [ME08]	Bąk et al. 2011 [BCW11]	Abele et al. 2010 [APS+10]	Classen et al. 2011 [CBH11]	Thüm et al. 2011 [TKES11]	Rosenmüller et al. 2011 [RSTS11]	Schroeter et al. 2012 [SLW12]	Kowal et al. 2013 [KSS13]	Acher et al. 2013 [ACLF13]	Seidl et al. 2014 [SSA14a]
Feature Modeling Notation	**Mandatory Features**	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Optional Features	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Feature Cardinality	-	-	-	-	-	-	o	+	+	-	-	-	+	o	+	+	+	-	-	o	-	-	o
	Attributes	-	-	+	-	-	-	o	+	+	-	-	+	-	-	+	+	+	-	-	-	-	-	-
	Feature Versions	-	-	-	-	-	-	-	-	-	-	-	-	-	o	-	-	-	-	-	-	-	-	+
	Layers	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	-	-	+	-
	External Features	-	-	-	-	-	+	-	-	-	-	-	-	-	-	+	+	+	-	+	-	-	+	(+)
	Binding Times	-	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	-
	Resource Mapping	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	-	+	+	-	-
	Alternative-Groups	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Or-Groups	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Group Cardinality	-	-	-	-	-	-	+	-	+	-	-	-	+	+	+	+	+	-	-	+	-	-	+
	Multiple Groups	(-)	(-)	(-)	(-)	+	+	+	+	+	-	(+)	+	(-)	(-)	+	+	(-)	-	+	(-)	(-)	(-)	+
Constraint Language	**Expressiveness**	R	R	R	R	O	R	O	O	-	P	R	-	P	R	P+	P	P+	P	P	R	P	P	P+
	Representation	T	T	G	G	T	G	G,T	T	-	T	G	-	T	T	T	G,T	T	T	T	G	T	T	T

R: Requires/Excludes, O: OCL, P: Propositional Logic, T: Textual, G: Graphical

Table 1: Distinctive characteristics of the inspected feature modeling notations.

3.2 Constraint Languages

The inspected feature modeling approaches utilize various constraint languages. They differ in their expressiveness and their representation as presented in the bottom part of Table 1.

Expressiveness of constraint languages is determined by the employed formalism and its utilized language constructs. For one, mere *requires* and *excludes* relations (R) may be specified [Ka90, GFA98, HSVM00, vGBS01, EBB05, SLW12]. Furthermore, the OCL (O) is used in some approaches [CE00, Cz02]. In addition, it is possible to utilize propositional logic (P). Depending on the concrete work, different subsets of Boolean operators are utilized to specify constraints over features [KSS13, SSA14a]. In addition, new language constructs are introduced for special purposes (P+), e.g., to compare attribute values [KOD10] or to express constraints over feature versions [SSA14a].

Representation of constraints is either graphical or textual. With a graphical (G) representation, additional edges are added between two features to express requires or excludes relationships [GFA98, HSVM00, vGBS01, EBB05, SLW12]. Textual representations (T) may be employed for a wider range of formalism, such as requires and excludes relationships [Ka90], OCL [CE00, Cz02] or subsets of propositional logic [KSS13, SSA14a]. In the latter case, there is a further distinction on how Boolean operators are represented as they may use logical symbols (e.g., $\wedge$, $\vee$), a verbal representation (e.g., *and*, *or*) or a representation known from various programming languages such as Java or C++ (e.g., `&&`, `||`). In some cases [Ri02, Ab10], both a textual and a graphical representation of constraints is provided.

4 Feature Modeling Family

From the results of the analysis in Section 3, we define a software family of feature modeling notations and constraint languages, which subsumes the individual approaches examined in Section 3. Furthermore, it is possible to generate variants with combinations of language constructs that currently do not exist in the literature or in practice.

We use a feature model to describe all valid configurations of the family in terms of configuration knowledge. Due to its size, the graphical representation of the feature model is split up over multiple figures. Figure 1 shows an overview of the top-level features of the family with `FeatureModel` describing the configuration options of the feature modeling notation and `ConstraintLanguage` capturing the configuration options for the cross-tree constraint language. Both these features are refined and described in detail in the following sections.

Figure 1: Top-level features of the feature modeling family. `FeatureModel` and `ConstraintLanguage` are defined in Figure 2 and Figure 3, respectively.

4.1 Variability of Feature Modeling Notations

Figure 2 shows a refined view of the feature model branch describing configuration options for the various feature modeling notations. We modeled both `Features` and `Groups` to be mandatory parts of each feature modeling notation. Features represent their type by either using a `FeatureCardinality` (e.g., [1..1] for mandatory) or an explicit `FeatureVariationType` (i.e., `OptionalFeatures` or `MandatoryFeatures`). When using feature cardinalities, it is further possible to allow `ClonedFeatures` if a feature can be instantiated multiple times. In addition, it is possible to explicitly allow `UnlimitedFeatures` by providing an unbounded maximum cardinality (using *) instead of an integer value. As the latter case implicitly depends on cloned features being enabled, we introduced constraint (1). Furthermore, it is possible to enable `External` to reference features defined in a different feature model, e.g., to realize feature layers [Ka98]. Using `Resources` allows association of features with arbitrary resources. Finally, enabling `BindingTimes` allows assigning a binding time to a feature, e.g., *compile time*.

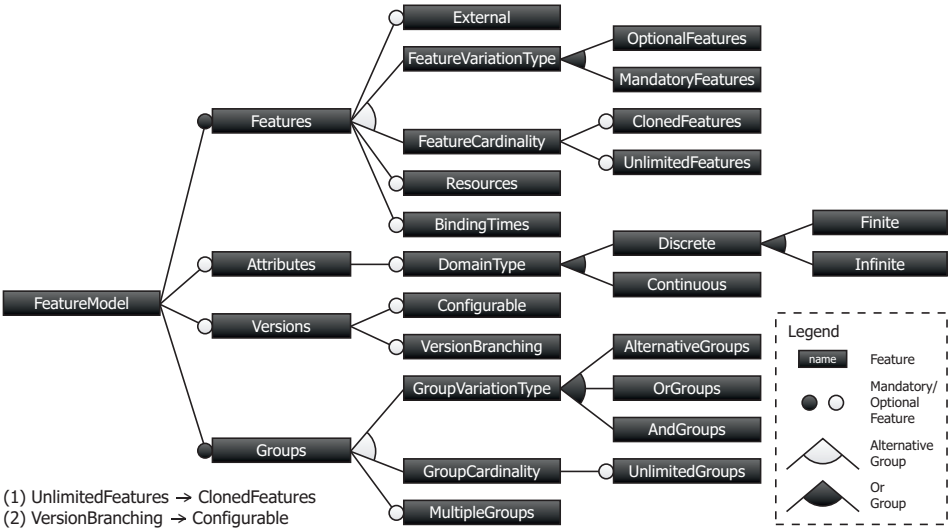


Figure 2: Feature model branch describing configuration options for feature modeling notations.

Furthermore, `Attributes` may be included in the feature model notation. Optionally, attributes have a `DomainType`, which specifies the domain for an attribute as either `Discrete` (e.g., enumerations, integer numbers or strings) or `Continuous` (e.g., floating point numbers). A discrete domain type may further be either `Finite` (e.g., enumerations) or `Infinite` (e.g., integer numbers or strings).

Additionally, it is possible to enable feature `Versions` to support variability in time. It can be decided whether versions can be used as `Configurable` units (as in [SSA14a]) or not (as in [ME08]). Versions are arranged along a chronological development line that is assumed to be linear unless `VersionBranching` is selected, which allows different branches, which depends on configurable versions so that we introduced constraint (2).

Similarly to features, groups represent their type either as `GroupCardinality` or as explicit `GroupVariationType`. To subsume the expressiveness of the inspected approaches, it would have been possible to model `AlternativeGroups` as a mandatory feature and `OrGroups` as an optional feature. We decided to use an or-group instead to give more liberty in variant derivation and also allow feature models that do not use alternative-groups. When using group cardinalities, it is possible to enable `UnlimitedGroups`, which permit an arbitrary number of features to be selected by providing an unbounded maximum cardinality (using *) instead of an integer value. Generally, only a single group is allowed as child of a feature unless `MultipleGroups` is selected.

4.2 Variability of Constraint Languages

Figure 3 shows a refined view of the feature model branch describing configuration options of the various cross-tree constraint languages. We designed the feature model of constraint languages to support `Propositional` logic or `OCL` as well as textual and graphical representations.

For propositional logic, various options for different language constructs exist (grouped by their respective number of operands). A number of `AtomicConstructs` is provided: The mandatory `FeaturePresence` checks whether a specified feature is part of a configuration. `AttributeRestrictions` allow comparison of attribute values using various arithmetic operators (`LT` ($<$), `LEQ` ($\leq$), `GEQ` ($\geq$) and `GT` ($>$)), string operators (`SUBS` as substring comparison) or those used in both contexts (`EQ` ($=$) and `NEQ` ($\neq$)).

Optionally, `VersionRestrictions` [SSA14a] may be selected with `VersionRangeRestrictions` allowing dependence on an interval of versions, `RelativeVersionRestrictions` specifying a valid set of versions in relation to a given version and `ConditionalVersionRestrictions` allowing evaluation of the former constructs for configurable versions only if their respective containing feature is present in a configuration. As restrictions for attributes and versions may only be specified if the respective elements are part of the notation, constraints (3) and (4) were added.

The sole child of `UnaryConstructs` is the feature `Not` ($\neg A$) as logical negation, which may be deselected by not selecting its parent. Furthermore, various `BinaryConstructs` are provided as known from Boolean algebra:

- `Or`: $A \vee B$ (logical or)
- `And`: $A \wedge B$ (logical and)
- `Xor`: $(A \oplus B) \equiv ((A \wedge \neg B) \vee (\neg A \wedge B))$
- `Implies`: $(A \rightarrow B) \equiv (\neg A \vee B)$[1]
- `Equivalent`: $(A \equiv B) \equiv ((A \rightarrow B) \wedge (B \rightarrow A))$
- `Excludes`: $(A$ `excludes` $B) \equiv \neg(A \wedge B)$

[1] We did not define a `Requires` feature as it is semantically equivalent to the already existing `Implies`.

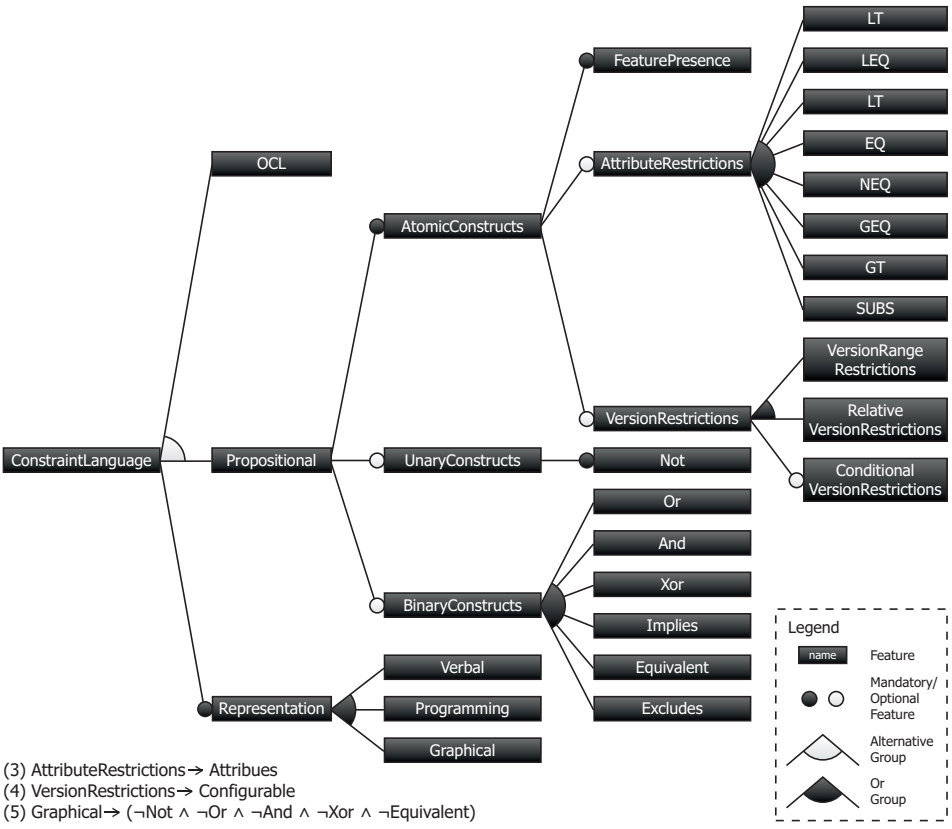


Figure 3: Feature model branch describing configuration options for cross-tree constraint languages.

The expressiveness of the constraint language may differ with the selection of supported constructs: For example, both selections {`Or`, `Not`} and {`Implies`, `Excludes`} result in a language that is complete with regard to Boolean logic. In contrast, when only selecting `Not` as construct, the expressiveness of the resulting cross-tree constraint language is severely limited.

It is possible to choose different representations for these constructs (`Representation`) as either textual or graphical: With a `Verbal` representation, textual literal names are used for constructs (e.g., *and*, *or*). With a `Programming` representation, textual operands similar to those used in Java or C++ are used (e.g., `&&`, `||`)[2]. Alternatively, it is also possible to choose a `Graphical` representation where constraints are added as additional edges between features to the visual representation of the feature model. This type of representation is only capable of visualizing implications and exclusions so that constraint (5) was added to exclude all other constructs when the graphical representation is selected. The type

[2] With the implementation of the feature modeling family in mind, we did not include a constraint representation that uses logical operators from Boolean logic (e.g., ∧, ∨) as those symbols cannot be typed on a keyboard.

of representation for constraints is further relevant for a generation of variants because a textual representation requires a grammar for the language to be supplied (see Section 5).

5 Case Study

To demonstrate the feasibility of our approach, we performed a case study using the presented family of feature modeling notations and constraint languages.[3] For this purpose, we provide a prototypical model-based realization of the aforementioned family of feature models and constraint languages in the form of an SPL as depicted in Figure 2 and Figure 3. This SPL may be used to generate variants of the underlying meta-models and grammars for individual feature model notations and cross-tree constraint languages regarding the described configuration options. Within the case study, we are particularly interested in answering two research questions:

RQ1 Is it possible to derive meta-models for feature models and constraint languages that are as expressive as the approaches analyzed in Section 3?

RQ2 Does the family of feature modeling notations support derivation of notations that have not been devised before?

We employ the transformational variability realization mechanism delta modeling [Sc10] to generate different variants of the family. In delta modeling, a *base variant* of a system is transformed to a *target variant* by applying a number of *delta modules* that each specify a set of coherent transformations defined as sequence of *delta operations*. A *delta language* provides the delta operations available to alter a *source language* by adding, modifying and removing elements. A variant is derived by selecting a valid subset of delta modules (e.g., using a feature model with a mapping to delta modules) and applying the delta modules in a suitable order to generate the intended target variant by transforming the base variant.

As base variant for the parts of the software family regarding the feature model and the constraint language based on propositional logic, we use the meta-models for Hyper Feature Models and their version-aware constraint language as used in our previous work [SSA14a, SSA14c], which provide language constructs as described by the last column of Table 1. Figure 4 depicts a representative excerpt of the base meta-model for feature models defined using Ecore[4] of the Eclipse Modeling Framework (EMF). Similarly, the base meta-model for constraint languages in propositional logic is also defined in EMF Ecore but further uses a concrete syntax to define a textual language using the tool EMFText[5]. For OCL, we use the meta-model and textual representation provided by the DresdenOCL toolkit[6], which is again based on EMF Ecore.

To make these artifacts subject to variability in delta modeling, we defined delta languages for both Ecore meta-models and concrete syntax files of EMFText using the delta language

[3] `https://fusionforge.zih.tu-dresden.de/projects/snowflake`
[4] `http://eclipse.org/emf`
[5] `http://emftext.org`
[6] `http://dresden-ocl.org`

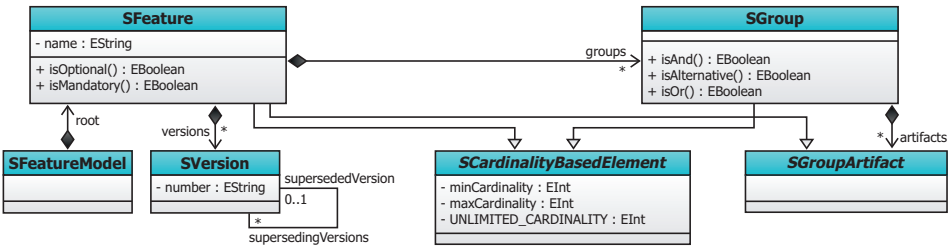


Figure 4: Excerpt from the base variant for the Ecore meta-model of the feature model family.

generation framework DeltaEcore[7] [SSA14b]. We further defined 58 delta modules that realize changes to accommodate for the different language constructs of the feature model and constraint language families when generating variants. We assigned delta modules to (combinations of) features of Figure 2 and Figure 3 so that variants can be generated by selecting a valid configuration of features, determining the respective delta modules and applying them in an automatically determined suitable order.

We defined configurations to represent the distinctive characteristics of each analyzed work by selecting features from our family of feature modeling notations that reflect the entries in each column of Table 1. We used these configurations to generate a variant for each inspected work consisting of the meta-models for the feature model and its constraint language as well as the concrete syntax file of the textual constraint language (if applicable). We inspected the generated variants for conformance with the selected configurations as well as their expressiveness with regard to the included distinctive characteristics. We used the generated variants of the meta-model and the constraint language to recreate the examples presented in each of the analyzed works[8]. Figure 5 shows an example of a variant of the meta-model for feature models for the configuration that consists of the features `FeatureModelingFamily`, `FeatureModel`, `Features`, `External`, `FeatureVariationType`, `OptionalFeatures`, `MandatoryFeatures`, `Groups`, `GroupVariationType`, `AlternativeGroups`, `OrGroups`, `AndGroups` and `MultipleGroups`. This variant resembles the notation used for the diagrams presented in Figure 1, Figure 2 and Figure 3.

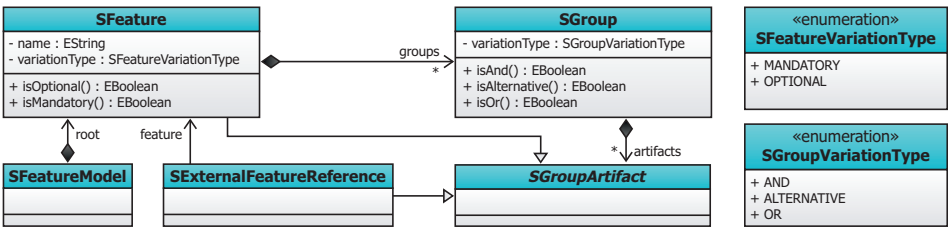


Figure 5: Excerpt from an example variant for the Ecore meta-model of the feature model family resembling the feature model notation used for the diagrams of this paper.

[7] `http://deltaecore.org`

[8] For papers, we recreated all presented examples. However, [CE00] is a book with over 800 pages so that we focused on creating a sample of all presented feature models.

As a result of generating specifically tailored variants of feature model notations and cross-tree constraint languages, the majority of concepts could be expressed directly by dedicated language constructs. In addition, we used external features linking different feature models to realize layers [Ka98]. However, we could not directly represent calculated attribute values [BTRC05] but had to substitute constraints on the attributes demanding respective values as a workaround. Finally, we had to create mock up models for the externally defined resources (e.g., hardware, views) to realize resource mappings [SLW12, KSS13], which are otherwise supplied along with a particular SPL. Using these techniques, we were able to capture all information presented in the original work by employing a variant generated from the feature modeling family. As a conclusion, we were able to answer *RQ1* positively as we could generate feature model notations and cross-tree constraint languages with similar expressiveness as the inspected approaches with regard to the information available in the respective publications.

In addition, we defined configurations for feature modeling notations that have a combination of language constructs that, to our knowledge, did not exist, yet. For example, we derived the variant for the feature models presented as diagrams in this paper, which includes multiple groups and external features (as presented in Figure 5). Furthermore, we generated previously non-existent variants, such as a variant with cardinality-based features as well as binding times and resource mappings for features. As a result, we were able to answer *RQ2* positively.

6 Conclusion

In this paper, we presented a family of feature modeling notations and constraint languages that encompasses various similar, yet different notations in order to generate specifically tailored variants of feature model and cross-tree constraint notations. As basis, we analyzed state of the art in feature modeling notations and classified 23 approaches by the notational concepts they offer. From these findings, we assembled conceptual configuration knowledge within feature models for a family of feature modeling notations and cross-tree constraint languages. We provided a model-based realization of this family and used it in a case study to demonstrate feasibility of our approach by generating variants with expressiveness similar to the analyzed approaches as well as previously non-existent notations. Using our work, it is possible to bootstrap SPL techniques to use them on feature models and their constraint languages to generate variants of feature model and cross-tree constraint notations according to a particular selection of language constructs. This is beneficial when depending on a notation with specific capabilities and may further be useful when transforming configuration knowledge specified in different feature modeling notations, analyzing configuration options of various different feature models or integrating configuration knowledge from various sources with different notations.

In our future work, we will extend the provided SPL with variability of analyses techniques, possible graphical representations, the generation of adequate tools for the variants as well as support for different SPL implementation techniques. A configuration of this feature modeling notation may determine which solvers (e.g., SAT, BDD, CSP) can be used, which

analysis techniques are available and also which other dependencies need to be considered for an implementation (e.g., if a chosen feature modeling notation restricts the choice for an SPL implementation techniques). Additionally, we will analyze effects of transforming models conforming to one variant of the feature model family to conform to another variant by substituting language constructs. As far as feasible, we will provide an implementation based on model transformation to allow data exchange between different variants of the family for feature modeling notations.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under grants SCHA1635/2-1 and SCHA1635/4-1 and by the European Commission within the project HyVar (grant agreement H2020-644298).

References

[Ab10] Abele, Andreas; Papadopoulos, Yiannis; Servat, David; Törngren, Martin; Weber, Matthias: The CVM Framework-A Prototype Tool for Compositional Variability Management. VaMoS, 10:101–105, 2010.

[Ac13] Acher, Mathieu; Collet, Philippe; Lahire, Philippe; France, Robert B.: FAMILIAR: A Domain-specific Language for Large Scale Management of Feature Models. Sci. Comput. Program., 78(6):657–681, June 2013.

[B3] Bürdek, Johannes; Lity, Sascha; Lochau, Malte; Berens, Markus; Goltz, Ursula; Schürr, Andy: Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '14, 2013.

[Ba05] Batory, D.: Feature Models, Grammars, and Propositional Formulas. Software Product Lines, 2005.

[BCW11] Bak, Kacper; Czarnecki, Krzysztof; Wasowski, Andrzej: Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In: Proceedings of the Third International Conference on Software Language Engineering. SLE'10, Springer-Verlag, Berlin, Heidelberg, pp. 102–122, 2011.

[BSRC10] Benavides, David; Segura, Sergio; Ruiz-Cortés, Antonio: Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems, 2010.

[BTRC05] Benavides, David; Trinidad, Pablo; Ruiz-Cortés, Antonio: Automated Reasoning on Feature Models. In: Proceedings of the 17th International Conference on Advanced Information Systems Engineering. CAiSE'05. Springer, 2005.

[CBH11] Classen, Andreas; Boucher, Quentin; Heymans, Patrick: A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. Science of Computer Programming, 2011.

[CE00] Czarnecki, Krzysztof; Eisenecker, Ulrich W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CHE04] Czarnecki, Krzysztof; Helsen, Simon; Eisenecker, Ulrich: Staged Configuration Using Feature Models. In (Nord, RobertL., ed.): Software Product Lines, volume 3154 of Lecture Notes in Computer Science, pp. 266–283. Springer Berlin Heidelberg, 2004.

[CHE05] Czarnecki, Krzysztof; Helsen, Simon; Eisenecker, Ulrich: Formalizing Cardinality-Based Feature Models and their Specialization. In: Software Process: Improvement and Practice. 2005.

[Cz02] Czarnecki, Krzysztof; Bednasch, Thomas; Unger, Peter; Eisenecker, Ulrich: Generative Programming for Embedded Software: An Industrial Experience Report. In (Batory, Don; Consel, Charles; Taha, Walid, eds): Generative Programming and Component Engineering, Lecture Notes in Computer Science. Springer, 2002.

[Dh13] Dhungana, Deepak; Seichter, Dominik; Botterweck, Goetz; Rabiser, Rick; Grünbacher, Paul; Benavides, David; Galindo, José A.: Integrating Heterogeneous Variability Modeling Approaches with Invar. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems. VaMoS '13, 2013.

[EBB05] Eriksson, Magnus; Börstler, Jürgen; Borg, Kjell: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In (Obbink, Henk; Pohl, Klaus, eds): Software Product Lines, volume 3714 of Lecture Notes in Computer Science, pp. 33–44. Springer Berlin Heidelberg, 2005.

[ES13] Eichelberger, Holger; Schmid, Klaus: A Systematic Analysis of Textual Variability Modeling Languages. In: Proceedings of the 17th International Software Product Line Conference. SPLC '13, ACM, New York, NY, USA, pp. 12–21, 2013.

[ES14] Eichelberger, Holger; Schmid, Klaus: Mapping the design-space of textual variability modeling languages: a refined analysis. International Journal on Software Tools for Technology Transfer, pp. 1–26, 2014.

[GFA98] Griss, M. L.; Favaro, J.; Alessandro, M. d': Integrating Feature Modeling with the RSEB. In: Proceedings of the 5th International Conference on Software Reuse. ICSR '98, IEEE Computer Society, Washington, DC, USA, 1998.

[HSVM00] Hein, Andreas; Schlick, Michael; Vinga-Martins, Renato: Applying Feature Models in Industrial Settings. In: Proceedings of the First Conference on Software Product Lines: Experience and Research Directions. Kluwer Academic Publishers, 2000.

[HTH13] Hubaux, Arnaud; Tun, Thein Than; Heymans, Patrick: Separation of Concerns in Feature Diagram Languages: A Systematic Survey. ACM Comput. Surv., 45(4):51:1–51:23, August 2013.

[Jé12] Jézéquel, Jean-Marc: Model-Driven Engineering for Software Product Lines. ISRN Software Engineering, 2012.

[Ka90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University SEI, 1990.

[Ka98] Kang, Kyo C.; Kim, Sajoong; Lee, Jaejoon; Kim, Kijoo; Shin, Euiseob; Huh, Moonhang: FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. Ann. Softw. Eng., 5:143–168, January 1998.

[Ka10] Kang, Kyo: , FODA: Twenty Years of Perspective on Feature Modeling. Keynote at the 4th International Workshop on Variability Modelling of Software-Intensive Systems, 2010.

[KOD10] Karataş, Ahmet Serkan; Oğuztüzün, Halit; Doğru, Ali: Global Constraints on Feature Models. In: Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming, CP'10, pp. 537–551. Springer-Verlag, Berlin, Heidelberg, 2010.

[KSS13] Kowal, Matthias; Schulze, Sandro; Schaefer, Ina: Towards Efficient SPL Testing by Variant Reduction. In: Proceedings of the 4th International Workshop on Variability & Composition. VariComp '13, ACM, New York, NY, USA, pp. 1–6, 2013.

[Li03] Lichter, Horst; von der Maßen, Thomas; Nyßen, Alexander; Weiler, Thomas: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung. Technical Report AIB-2003-07, Department of Computer Science, RWTH Aachen, July 2003.

[ME08] Mitschke, R.; Eichberg, M.: Supporting the Evolution of Software Product Lines. In: ECMDA Traceability Workshop. ECMA-TW, 2008.

[Ri02] Riebisch, M.; Böllert, K.; Streitferdt, D.; Philippow, I.: Extending Feature Diagrams with UML Multiplicities. In: 6th World Conference on Integrated Design & Process Technology (IDPT2002). June 2002.

[Ro11] Rosenmüller, Marko; Siegmund, Norbert; Thüm, Thomas; Saake, Gunter: Multi-dimensional Variability Modeling. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. VaMoS '11, ACM, New York, NY, USA, pp. 11–20, 2011.

[Sc07] Schobbens, Pierre-Yves; Heymans, Patrick; Trigaux, Jean-Christophe; Bontemps, Yves: Generic Semantics of Feature Diagrams. Comput. Netw., 51(2):456–479, February 2007.

[Sc10] Schaefer, Ina; Bettini, Lorenzo; Bono, Viviana; Damiani, Ferruccio; Tanzarella, Nico: Delta-Oriented Programming of Software Product Lines. In: Software Product Lines: Going Beyond, pp. 77–91. Springer, 2010.

[SHT06] Schobbens, Pierre-Yves; Heymans, Patrick; Trigaux, Jean-Christophe: Feature Diagrams: A Survey and a Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference. RE '06, IEEE, Washington, DC, USA, pp. 136–145, 2006.

[SLW12] Schroeter, Julia; Lochau, Malte; Winkelmann, Tim: Multi-Perspectives on Feature Models. In: Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, 2012.

[SSA14a] Seidl, Christoph; Schaefer, Ina; Aßmann, Uwe: Capturing Variability in Space and Time with Hyper Feature Models. In: Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). VaMoS'14, 2014.

[SSA14b] Seidl, Christoph; Schaefer, Ina; Aßmann, Uwe: DeltaEcore-A Model-Based Delta Language Generation Framework. In: Modellierung. Modellierung'14, 2014.

[SSA14c] Seidl, Christoph; Schaefer, Ina; Aßmann, Uwe: Integrated Management of Variability in Space and Time in Software Families. In: Proceedings of the 18th International Software Product Line Conference (SPLC). SPLC'14, 2014.

[Th11] Thüm, T.; Kästner, C.; Erdweg, S.; Siegmund, N.: Abstract Features in Feature Modeling. In: 15th International Software Product Line Conference (SPLC). 2011.

[vGBS01] van Gurp, J.; Bosch, J.; Svahnberg, M.: On the Notion of Variability in Software Product Lines. In: Proceedings of the Conference on Software Architecture. 2001.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 173

Konzeptionelle Modellierung ausführbarer Event Processing Networks für das Event-driven Business Process Management

Stefan Gabriel[1], Christian Janiesch[2]

Abstract: Unternehmensübergreifende Geschäftsprozesse müssen nicht nur interne, sondern auch externe Ereignisse berücksichtigen, um adäquat auf auftretende Situationen reagieren zu können. Eine kurze Reaktionszeit auf Basis von Event-driven BPM-Systemen verbessert hierbei den Entscheidungsspielraum. Die Planung derartiger Systeme auf Basis von BPM- und CEP-Technologie ist derzeit allerdings nicht ohne frühzeitige Festlegung auf proprietäre Technologie möglich. Um dem zu begegnen, schlagen wir eine Sprache und Architektur zur konzeptionellen Modellierung und Serialisierung in ausführbaren Code von CEP-Modellen für das Event-driven BPM vor. Wir zeigen, wie eine entsprechende Implementierung auf Basis einer herstellerunabhängigen CEP-Modellierungssprache und Notation, umgesetzt in der Open-Source-Modellierungs-Plattform Oryx, für die Esper Event Processing Language aussehen kann und demonstrieren diese an einem Beispiel.

Keywords: Event-driven Business Process Management, Complex Event Processing, Complex Event Processing Model & Notation, Model-driven Engineering.

1 Einleitung

Die Maßnahmen und Entscheidungen zur Reaktion auf auftretende Geschäftsvorfälle werden insbesondere durch die Latenzzeiten in der Datenübermittlung und Datenverarbeitung beeinflusst. Je mehr Zeit zwischen dem Auftreten der geschäftsrelevanten Ereignisse und der getroffenen Maßnahme vergeht, desto niedriger ist in der Regel der geschäftliche Nutzen der Entscheidung [MS10; OJ15]. Jedes Unternehmen sollte daher bestrebt sein, seine Geschäftswerte zu optimieren und die Latenzzeiten zu reduzieren. Aus Sicht der IT lässt sich dies bspw. durch den Einsatz ereignisgesteuerter Technologien im Sinne eines Business Activity Monitoring (BAM) [Ga02] oder weiterführend eines Event-driven Business Process Management (EdBPM) [Ja12, Kr14] reduzieren.

Existierende Lösungen haben aber gemein, dass sie jeweils nur einen Ausschnitt des Gesamtproblems lösen. Wesentliche Faktoren stellen Hindernisse für die Benutzung solcher Lösungen dar. Zu diesen Faktoren zählen: fehlende offene Standards [BD10], Abhängigkeit vom eingesetzten (rudimentären) Produkt [Ja12] und die Abstraktion zwischen Modellierung und technischer Umsetzung [Du13].

[1] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, stefan-gabriel@web.de
[2] Julius-Maximilians-Universität Würzburg, Juniorprofessur für Information Management, Sanderring 2, 97070 Würzburg, christian.janiesch@uni-wuerzburg.de

Für die Entwicklung von Complex Event Processing (CEP)- und EdBPM-Systemen sind weitere Ansätze bereits vorhanden: [EN11] beschreiben eine grundlegende Idee für die semantische Benutzung ereignisverarbeitender Konstrukte. [Fr12] haben dies graphisch und in Anlehnung an BPMN [OM13] für die Anwendung im BAM umgesetzt. [BD10] erläutern den theoretischen Aufbau und praktische Herangehensweise zur Einführung ereignisgesteuerter Systeme in Organisationen. Verwandte Arbeiten im Bereich der Integration von BPM(N) und CEP finden sich auch bei [De07; Ku10]. Hier geht es aber weniger um CEP im engeren Sinne als um die Verbesserung des Event-Handlings in BPMN. [Vi14] schlägt die Event Processing Model and Notation ([moby-]EPMN) vor. Es handelt sich allerdings nicht um eine geeignete Methode zur konzeptionellem, semantischen Modellierung von Event Processing Networks (EPN). Die Notation dient im Wesentlichen dazu, Konjunktion, Disjunktion, Sequenzen, und funktionale Annotationen vorzunehmen. Eine umfangreiche Übersicht zum EdBPM findet sich auch bei [Kr14].

Im vorliegenden Beitrag stellen wir eine Software-Architektur als Basis für das EdBPM vor, die es erlaubt konzeptionell und anbieterunabhängig EPN für das CEP zu modellieren und diese automatisch in entsprechenden CEP-Code zu transformieren. Wir haben dafür die Vorabreiten von [Fr12] auf CEP erweitert und eine Transformationsschicht geschaffen, die diese anbieterunabhängigen Modelle beispielhaft in Esper-Code übersetzt, der dann automatisiert zum Monitoring von Prozessen eingesetzt werden kann.

2 Konzeptionelle Modellierung von Event Processing Networks

Die grundlegende Idee zum zugrundeliegenden Meta-Modell der Modellierungssprache basiert auf [Fr12]. Die nachfolgende Spezifikation des Meta-Modells (siehe Abb. 1), die Notation und zugehörige Semantiken, Umsetzung in einer Programmiersprache und Serialisierung sind eine Weiterentwicklung dieser Grundidee.

Die Modellierung eines *EventProcessingNetwork* entspricht der Modellierung eines EPN. Ein Modell wird sequentiell erstellt und die einzelnen Ereignisse gelangen ebenso sequentiell in die entsprechenden *EventStreams*.

Alle Knoten werden über die abstrakte Klasse *FlowNode* weiter verfeinert und entsprechend alle Kanten über die abstrakte Klasse *Edge*. Als Flussknoten stehen die Adapter und alle Event Processing Agents (EPA) zur Verfügung. Diese Arten werden als abstrakte Klasse definiert. Die Klasse *EventProcessingAgent* hat drei Spezialsierungen mit weiteren Konkretisierungen basierend auf [EN11]: die konkrete Klasse *Filter*, die abstrakten Klassen *Transformation* und *PatternDetect*. Die Klasse Transformation wird von den EPAs *Aggregate*, *Translate*, *Split* und *Compose* konkretisiert. Die Klasse *Enrich* ist ein Sonderfall von Translate und erbt daher von dieser Klasse. Ein Objekt der Klasse PatternDetect beinhaltet genau ein Objekt der Klasse *Pattern*. Die Klasse Pattern spezialisiert elementare Muster durch die abstrakte Klasse *BasicPattern* und dimensionale Muster durch die abstrakte Klasse *DimensionalPattern*. Der Filter-EPA kann neben dem Filtern von Informationen auch Ereignisse projizieren. Dabei kann ein Filter-Objekt

keine bis alle Ereignisattribute projizieren und definiert dadurch einen eigenen Ereignistyp für den ausgehenden EventStream.

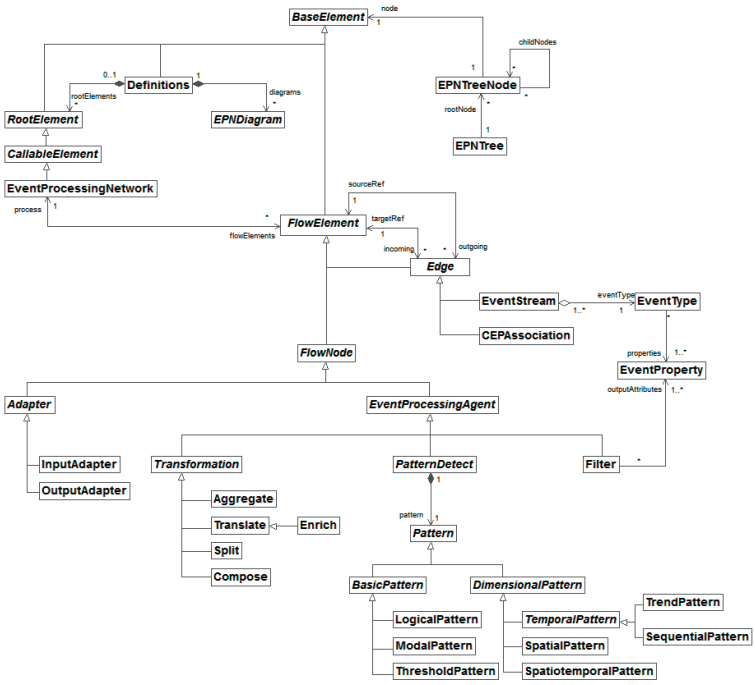


Abb. 1: Complex Event Processing Model & Notation Meta-Modell (Ausschnitt)

Die Modellierungssprache kann an BPMN angedockt werden, um EdBPM-Prozesse zu modellieren. Die folgenden wesentlichen Notationselemente in Abb. 2 geben einen Überblick darüber, wie Ereignisse verteilt, verarbeitet und benutzt werden, um Bezüge zwischen Ereignissen herzuleiten.

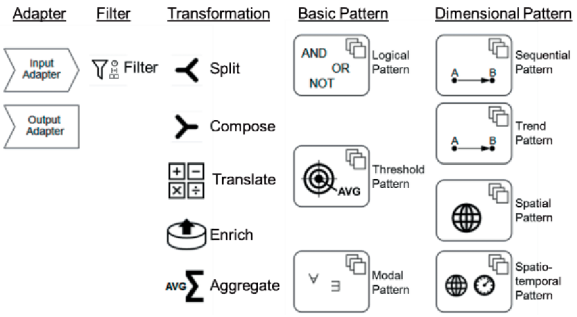


Abb. 2: Notationsübersicht (Ausschnitt)

3 Softwarearchitektur und Implementierung

Die Implementierung der Systemarchitektur folgt der üblichen Dreiteilung im CEP von Producer, Processor und Consumer [Lu02; EN11]. Wir haben dazu drei Komponenten implementiert: Die CEP-Applikation, die Event-Kommunikation und ein Modellierungswerkzeug.

Die CEP-Applikation ist der Kern der entwickelten Software und bindet die CEP-Engine und weitere Werkzeuge an. Das Projekt ist die CEP-Applikation und beinhaltet den Web Service zur Registrierung von Statements in der Esper Event Processing Language (EPL), implementiert die Schnittstellendefinition und enthält die Bereitstellungslogik des Registrierungs-Web-Services. Als CEP-Engine kommt hier Esper 5.1 zum Einsatz. Weiterhin wird eine MySQL-Datenbank betrieben, mit einem entsprechenden Connector an die Komponente angebunden und über eine eigene entwickelte Komponente zur Event-Kommunikation gekoppelt. Weiterhin beinhaltet die Applikation die Schnittstellen zu den Web Services zur Registrierung der EPL-Statements und zur Bereitstellung der Key Performance Indicators (KPI). Die Schnittstelle bietet Operationen an, um EPL-Statements an der CEP-Engine als einzelne oder gebündelte Anweisungen zu registrieren, und deren erzeugten KPI-Web-Services zu identifizieren und deren Daten bereitzustellen. Die Schnittstelle definiert wie der KPI-Web-Service seinen Wert bereitstellt.

Die Event-Kommunikation ist für die Ereignisübertragung zwischen den Ereignisproduzenten und der Ereignisverarbeitung zuständig. Sie enthält die nachrichtenorientierte Middleware (MOM) und implementiert das Publish-Subscribe-Entwurfsmuster als Event Channel. Die Komponenten benutzt die Tools Apache ActiveMQ, Apache CXF WS-Notification und Simple Logging Facade for Java (SLF4J). Als MOM wird Apache ActiveMQ eingesetzt. Die Implementierung des Publish-Subscribe-Entwurfsmuster von Apache CXF WS-Notification wurde für den speziellen Einsatz für die Software erweitert. Die MOM dient dem Entwurfsmuster zur Nachrichtenübermittlung. Das SLF4J ist ein Hilfstool für die Middleware, die Funktionalität zum Logging beinhaltet.

Als Modellierungswerkzeug wird der Oryx-Editor eingesetzt [Or12]. Der Oryx-Editor dient als Oberfläche einer ganzheitlichen Lösung, die sich von der Modellierung entsprechender Prozesse über die Serialisierung in EPL-Statements bis zur Veröffentlichung von KPIs als Web Services erstreckt. Grundsätzlich besteht der Oryx-Editor aus einem Frontend und Backend, die beide als Applikation auf einem Web-Server ausgeführt werden. Das Frontend ist über einen Browser abrufbar und bedienbar. Die Komponente benutzt die Tools Apache Tomcat 7 als Webserver, PostgreSQL Database als Datenbankserver für den Oryx-Editor und unsere CEP-Applikation. Der Web-Server und der Datenbankserver sind die Grundvoraussetzung für den Betrieb des Oryx-Editors. Das Stencil Set ist der Datensatz für die Benutzung der Modellierungssprache im Editor. Ein Stencil Set besteht aus der grafischen Darstellung der Sprachelemente, einer konkreten Beschreibung der Elemente mit Eigenschaften und Eingabemöglichkeiten und der Regeldefinition für die Sprache [Pe07]. Hierdurch wird das Meta-Modell syntaktisch als auch semantisch implementiert.

4 Serialisierung der konzeptionellen Modelle

Im Backend ist das Meta-Modell implementiert und alle Flusselemente werden als Klassen abgebildet. Diese Klassenstruktur ist eine allgemeine Zwischenstruktur des Diagramms, bevor es in die speziellen Elemente des Meta-Modells übersetzt wird. Factory-Klassen erzeugen die entsprechenden Objekte der implementierten Klassenstruktur anhand der Diagrammrepräsentation. Spezielle Klassen organisieren die Serialisierung eines Modell-Elements für die konkrete EPL der CEP-Engine Esper.

Das Frontend speichert die Diagramme anhand der Definition des entsprechenden Stencil Sets ab. Die Datenrepräsentation wird im Backend für die Konvertierung in die interne Struktur benötigt. Der Austausch der Daten zwischen Frontend und Backend erfolgt über ein Servlet. Es steuert die Konvertierung in die interne Struktur an und übersetzt die interne Struktur der Meta-Modelle in EPL. Weiterhin tritt das Servlet als Vermittler zwischen dem Frontend und Registrierungsservice der CEP-Applikation auf.

Nachdem die Modelle erzeugt wurden, müssen die Anweisungen zur Ereignisverarbeitung aus dem Modell zur CEP-Engine gelangen. Abb. 3 zeigt den Ablauf der Daten von der Modellierung bis zur Registrierung in der Engine.

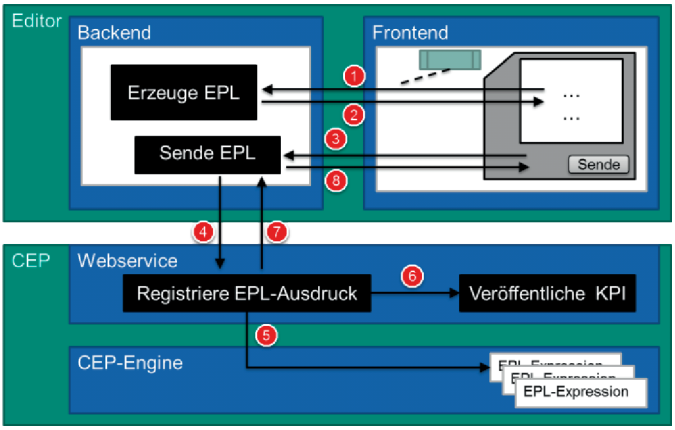


Abb. 3: Datenaustausch zwischen den Softwarekomponenten

Im Frontend des Oryx-Editors wird das entsprechende Plug-In ausgeführt und öffnet ein Dialogfenster zum Generieren der EPL. Dieses schickt die Diagrammdaten an das ein Servlet im Backend (1). Dort werden die Daten konvertiert und in entsprechende EPL-Statements übersetzt (2). Die EPL-Statements werden an das Servlet im Backend geschickt (3). Das Servlet sendet die Statements weiter an den Registrierungs-Web-Service der CEP-Applikation (4). Die Statements werden anschließend in der CEP-Engine registriert (5) und durch die modellierten KPIs als Web Service veröffentlicht (6). Danach sendet der Registrierungs-Web-Service die Adressen der erzeugten KPI-Web-Services an das Servlet zurück (7), der sie ans Dialogfenster im Frontend weiterleitet (8).

5 Implementierungsbeispiel

Das folgende Szenario beschreibt einen Kühllager-Monitoring-Prozess bei dem die Kühllagertemperatur beobachtet und bei Unregelmäßigkeiten reagiert wird (siehe Abb. 4). Die Schnelligkeit der Reaktion ist von Bedeutung, da Güter in Kühllagern verderblich sind.

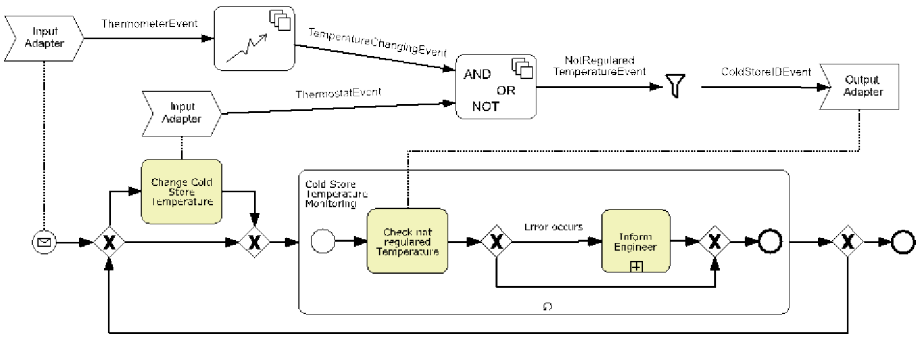


Abb. 4: Prozess *Kühllager-Monitoring* in BPMN mit CEPMN-Ergänzungen

Die Ereignisverarbeitung läuft wie folgt ab: Das Ereignis *Temperatur* des Typs *ThermometerEvent* geht in das System ein. Das *Trend*-Pattern untersucht die Temperaturereignisse auf eine streng steigende Temperatur. Ist dies der Fall, wird das letzte Ereignis des Trends als Ereignistyp *TemperatureChangingEvent* an ein *Logical*-Pattern weitergeleitet. Liegt diesem Pattern kein Ereignis *Thermostatregulierung* vom Typ *ThermostatEvent* vor, erzeugt es innerhalb des Zeitfensters ein Ereignis zur nichtregulierten Temperaturänderung vom Typ *NotRegularedTemperatureEvent*. Zum Schluss wird das Event auf die Eigenschaft *ColdStoreID* reduziert. Diese Eigenschaft ist der Wert der resultierenden KPI. Der Output-Adapter benennt mit seinem Namen den zugehörigen KPI-Web-Service. Parallel dazu wird ein Kühllager-Monitoring-Prozess in BPMN dargestellt, der durch die Echtzeitauswertungen über eine konkrete Aktivität beeinflusst werden kann. So wird deutlich an welcher Stelle im Prozess Daten ausgelesen und verwendet werden.

Die Serialisierung der einzelnen EPA und die Adapter werden in der Esper EPL generiert. Ein EPN beginnt mit mindestens einem Input- und endet mit mindestens einem Output-Adapter. Die Anweisung `INSERT INTO` gibt an, dass die Ergebnisse in einen EventStream weitergeleitet werden, der gleichzeitig neu erzeugt wird. Auf diese Weise wird der korrekte Event-Fluss im EPN zwischen den verschiedenen EPA sichergestellt.

Die Syntax `SELECT`, `FROM` und `WHERE` ist in der EPL gleichbedeutend mit den Konstrukten aus SQL. Über die Angabe `as` wird eine ausgehende Event-Eigenschaft benannt. Die Trend- und Logical-Pattern wenden eine Mustererkennung von Ereignissen und lösen dann ein Ereignis aus, wenn in ihren angegebenen Zeitfenstern die definierten Bedingungen erfüllt sind. Im Logical-Pattern heißen die Variablen gleich den Ereignis-

typen, die jeweils nach EventStreams benannt sind. Die Serialisierung hat folgende Form:

```
// this is an InputAdapter
INSERT INTO a8db2f2b
SELECT * FROM ThermometerEvent

// this is a TrendPattern
INSERT INTO 2705dfd2
SELECT {a.ColdStoreID, b.ColdStoreID} as ColdStoreID,
       {a.Temperature, b.Temperature} as Temperature
FROM pattern[every a=a8db2f2b ->
                  b=a8db2f2b(Temperature > a.Temperature)].win:
                                          time(20 second) output last

// this is an InputAdapter
INSERT INTO b9f185a9
SELECT * FROM ThermostatEvent

// this is a LogicalPattern
INSERT INTO a93f9b48
SELECT b9f185a9.EmployeeID as EmployeeID,
       b9f185a9.ColdStoreID as ColdStoreID,
       b9f185a9.Temperature as Temperature
FROM pattern[every 2705dfd2=2705dfd2 AND NOT b9f185a9=b9f185a9
            (ColdStoreID=2705dfd2.ColdStoreID)].win:time(30 minute)

// this is a Filter
INSERT INTO 9071b7fc
SELECT ColdStoreId as ColdStoreId
FROM a93f9b48

// this is an OutputAdapter
SELECT ColdStoreID as 9071b7fc
FROM 9071b7fc
```

6 Zusammenfassung und Ausblick

Existierende Produkte für das Echtzeit-Monitoring und -Management von Geschäftsprozessen bieten derzeit keinen offenen Ansatz, der die Planung und Entwicklung von EdBPM-Prozessen ohne die vorgelagerte Entscheidung für einen Anbieter proprietärer Software abbildet. Wir schlagen daher eine anbieterunabhängige Modellierung von EdBPM vor. Das Meta-Modell und die grafische Notation orientieren sich am Standard BPMN 2.0. Wir haben dabei sichergestellt, dass die Modellierung nicht nur ein Selbstzweck ist, sondern legen darüber hinaus dar, wie eine Transformation und Serialisierung vom Modell zu einem ausführbaren CEP-Statement möglich ist. Wir zeigen dies am Beispiel der Event-Verarbeitungssprache Esper EPL und der offenen CEP-Engine Esper und illustrieren das Ergebnis an einem Anwendungsbeispiel. Neben der Modellierung serialisiert die Software die resultierenden Modelle automatisiert in EPL-Statements und registriert diese in der CEP-Engine. Die spezifizierten KPI werden so automatisch und in Echtzeit über Web Services verfügbar. Während die gegenwärtige Implementierung auf

Esper basiert, ist es aber auch denkbar, vergleichbare auf SQL basierende CEP-Sprachen wie die Continuous Computation Language (CCL) oder Continuous Query Language (CQL) zu nutzen.

Literaturverzeichnis

[BD10] Bruns, R.; Dunkel, J.: Event-Driven Architectur. Springer, Berlin, 2010.

[De07] Decker, G.; Grosskopf, A.; Barros, A.: A Graphical Notation for Modeling Complex Events in Business Processes. In Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Annapolis, MD, 2007, S. 27.

[Du13] Dumas, M.; La Rosa, M.; Mendling, J.; Reijers, H.: Fundamentals of Business Process Management. Springer, Heidelberg, 2013.

[EN11] Etzion, O.; Niblett, P.: Event Processing in Action. Manning Publications, Cincinnati, OH, 2010.

[Fr12] Friedenstab, J.-P.; Janiesch, C.; Matzner, M.; Müller, O.: Extending BPMN for Business Activity Monitoring. In Proceedings of the 45th Hawai'i International Conference on System Sciences (HICSS), Maui, HI, 2012, S. 4158-4167.

[Ga02] Gartner Inc.: Business Activity Monitoring: Calm Before the Storm, 2002, http://www.gartner.com/resources/105500/105562/105562.pdf, Abruf am 21.10.2015.

[Ja12] Janiesch, C.; Matzner, M.; Müller, O.: Beyond Process Monitoring: A Proof-of-Concept of Event-driven Business Activity Management. Business Process Management Journal, 18 (4) 2012, S. 625-643.

[Kr14] Krumeich, J.; Weis, B.; Werth, D.; Loos, P.: Event-Driven Business Process Management: Where are we now? A Comprehensive Synthesis and Analysis of Literature. Business Process Management Journal, 20 (4) 2014, S. 615-633.

[Ku10] Kunz, S.; Fickinger, T.; Prescher, J.; Spengler, K.: Managing Complex Event Processes with Business Process Modeling Notation. In Proceedings of the 2nd International Workshop on BPMN. LNBIP Vol. 67, Potsdam, 2010, S. 78-90.

[Lu02] Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional, Boston, MA, 2002.

[MS10] zur Muehlen, M.; Shapiro, R.: Business Process Analytics. In Handbook on Business Process Management: Strategic Alignment, Governance, People and Culture. Vol. 2 (vom Brocke, J.; Rosemann, M., Eds.), Springer, Berlin, 2010.

[OJ15] Olsson, L.; Janiesch, C.: Real-time Business Intelligence und Action Distance: Ein konzeptionelles Framework zur Auswahl von BI-Software. In Proceedings of the 12. Internationale Tagung Wirtschaftsinformatik (WI), Osnabrück, 2015, S. 691-705.

[OM13] Object Management Group Inc.: Business Process Model and Notation (BPMN) Version 2.0.2, 2013, http://www.omg.org/spec/BPMN/2.0.2/PDF, Abruf am 18.06.2015.

[Or12] Oryx Editor. https://code.google.com/p/oryx-editor/, Abruf am: 21.10.2015.

[Pe07] Peters, N.: Oryx. Stencil Set Specification, https://oryx-editor.googlecode.com/files/OryxSSS.pdf, Abruf am 21.10.2015.

[Vi14] Vidačković, K.: Eine Methode zur Entwicklung dynamischer Geschäftsprozesse auf Basis von Ereignisverarbeitung. Dissertation. Universität Stuttgart, Stuttgart, 2014.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 181

BPM Considered Harmful

Stefan Berner[1]

Abstract: Business Process Modeling (BPM) spielt eine grosse Rolle in den frühen Schritten der Softwareentwicklung. Eine zu grosse Rolle nach Meinung des Autors. Fast alle Vorgehensmethoden empfehlen für den Beginn einer Geschäftsmodellierung BPM. Fast alle Business-Analysten nutzen BPM als Einstieg und einfachsten Zugang zu Wissen und Verständnis um eine Geschäftswelt.

Ohne gemeinsames Verständnis der Begriffe und ihrer Relevanz für eine Umgebung können weder künftige Nutzende noch Informatiker gute Geschäftsprozesse modellieren. BPM ist nicht der beste Weg dieses gemeinsame Verständnis zu erlangen. BPM hat weitere gravierende Nachteile, die es als Einstiegsvorgehen in ein IT-Projekt ungeeignet machen.

Dieses Paper zeigt die Nachteile des BPM auf. Es zeigt weiter auf, wie vor der Prozessmodellierung ein besseres Verständnis der zu modellierenden Welt erlangt werden kann.

Keywords: BPM, Prozessmodell, Informationsmodell, Softwarequalität

1 Einleitung

Mit seinem berühmten *letter to the editor* "Go To Statement Considered Harmful" [Di68] lancierte Dijkstra eine kontroverse Diskussion, die zur Akzeptanz strukturierter Programmierung führte. In diesem Sinne wurde der Ausdruck im Titel dieses Papers übernommen. Die vorliegenden Betrachtungen aus Sicht eines Praktikers sollen die Diskussion anstossen. Welche Rolle soll Business Process Modeling in einem Softwareprojekt spielen? Ist es wirklich der beste Einstieg und Zugang zu einer Softwarelösung?

BPM weist einige Nachteile auf. Sein typischer Einsatz führt nicht zu bester Softwarequalität. Die aufgeführten Beobachtungen und daraus abgeleiteten Hypothesen haben sich in der jahrelangen praktischen Projektarbeit des Autors bestätigt. Ein unabhängiger Nachweis liegt ausserhalb meiner Möglichkeiten. Dieser Beitrag soll Leute aus der Forschung anregen, die Aussagen und Hypothesen in neutralen Studien zu überprüfen.

[1] Diso AG, Morgenstrasse 1, CH-3073 Bern-Gümligen, sberner@diso.ch

2 Begriffe

2.1 Business Process Model

Das Business Process Model (Geschäftsmodell) dient dem Zweck, Geschäftsabläufe in einer für die Auftraggeber verständlichen Art zu beschreiben. Es soll gleichzeitig eine formale Vorlage für die Umsetzung in Software sein. Typischerweise sind die folgenden Aspekte in einem Business Process Model vereinigt:

Funktionen	Daten, Informationen
Ereignisse	Zustände
Organisationseinheiten	Akteure
Transportkanäle	Medien
Datenspeicher und -kanäle	Schnittstellen zu Umsystemen

Das Business Process Model beschreibt aus fachlicher Sicht was Akteure wie mit Informationen *tun*. Es beschreibt Geschäftsabläufe aus Sicht der Nutzenden. Beispiel:

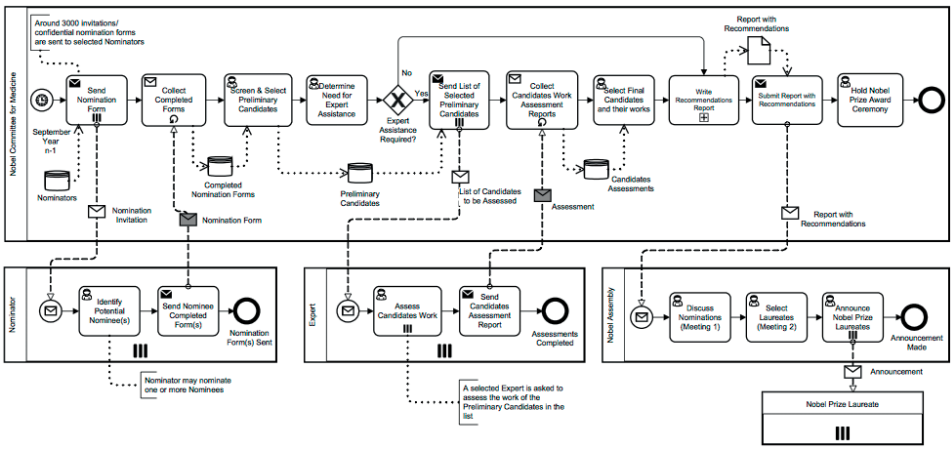


Abb. 1: Beispiel eines Business Process Models (siehe [BP10])

2.2 Informationsmodell

Ein Informationsmodell ist eine Beschreibung der in einer Umgebung benötigten Informationen. Es beschreibt Entitäten mit ihren Attributen und ihren Beziehungen zu anderen Entitäten. Es wird normalerweise als konzeptionelles Datenmodell bezeichnet. Der Begriff Informationsmodell wird verwendet um folgende Unterschiede zu einem Datenmodell hervorzuheben:

- Sämtliche Namen (Entitäten, Attribute, Beziehungen) sind ausschliesslich Begriffe der Geschäftswelt und werden von allen Beteiligten vorbehaltlos akzeptiert.

- Sämtliche Verknüpfungen von Entitäten sind in beide Richtungen mit einem Verb beschrieben.
- Es kommen keine technischen Begriffe (wie Datenbank, Tabellen, data types, keys, constraints, XML etc.) vor.

Das Informationsmodell beschreibt, was Dinge (Entitäten) fachlich miteinander *zu tun haben*. Wie sie strukturell zusammenhängen. Es beschreibt die möglichen, konsistenten und relevanten Zustände, die statische Struktur eines Systems.

Der Begriff *Datenmodell* wird im Zusammenhang mit Geschäftsmodellierung bewusst vermieden. Daten bilden Informationen in speicherbare, technisch verwaltbare Elemente ab. Datenwerte sind per se sinnlos. Anwenderinnen und Anwender nutzen nie Datenwerte. Sie nutzen Informationen, die durch Interpretation der Datenwerte entstehen. Das Informationsmodell ist die explizite Dokumentation der verbindlichen, fachlichen Interpretation der Daten einer Umgebung.

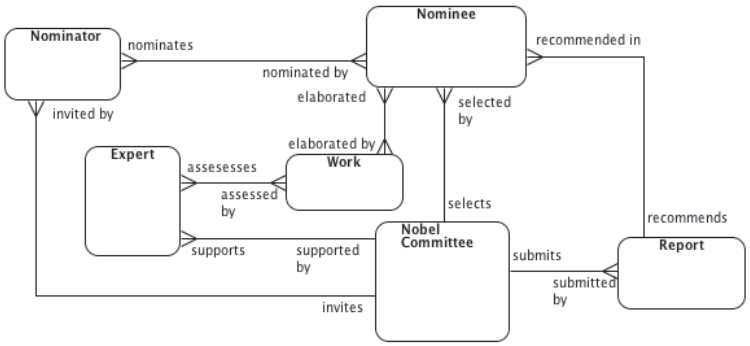


Abb. 2: Beispiel eines Informationsmodells

3 Kritik am BPM

3.1 Falsche Reihenfolge

Ein Prozess überführt ein System von einem konsistenten Zustand in einen anderen. Prozesse sind nie das Ziel, sie sind immer der Weg. Prozesse beschreiben, *wie* eine Zustandsänderung herbeigeführt wird. Sie beschreiben nicht die Zustände als solche. Das Business Process Model beschreibt was Akteure wie mit den Dingen (Entitäten) *tun*. Es beschreibt, wie durch eine Aktivität, unter Einsatz von (technischen) Hilfsmitteln, das System von einem konsistenten Zustand in einen anderen überführt wird.

Damit ein guter Prozess entworfen werden kann müssen Start und Ziel bekannt sein. Einerseits sind das Auslöser (Ereignisse) und der Anfangszustand, andererseits der erwartete

Endzustand der manipulierten Informationen. Ohne zu wissen, was das Ziel eines Prozesses ist, macht es wenig Sinn, diesen zu beschreiben. Konsistente Zustände werden durch das fachliche, statische Systemwissen – dem oben genannten Informationsmodell – beschrieben.

In der Praxis wird als erster Schritt einer Systembeschreibung meistens ein Prozessmodell erstellt. Warum das so ist wird weiter unten erläutert. Die darin verwendeten oder erzeugten Informationen werden in einem zweiten Schritt zu einem Datenmodell zusammengefasst. Eine vollständige, exakte Beschreibung der Informationsstruktur aus Sicht der Anwendenden und losgelöst von den Prozessen, ist in der Praxis kaum je anzutreffen. Diese fehlende Sicht auf die Begriffe und das mangelhafte Verständnis der Geschäftswelt führen i.d.R. dazu, dass Datenmodelle zu stark die Struktur der Prozesse und die Sichtweise der Entwickler abbilden. Strukturelle fachliche Probleme und Inkonsistenzen werden mit diesem Vorgehen erst spät im Entwicklungsprozess (bei der Datenmodellumsetzung, der Modulerstellung oder im Test) erkannt. Weil die Prozesse zu diesem Zeitpunkt in der Regel bereits abgenommen und fixiert sind, will man diese nicht mehr ändern. Das führt dazu, dass Architekturprobleme auf Ebene der Daten und Programme "ausgeglichen" werden. Dieses Vorgehen ist mit Sicherheit suboptimal.

3.2 Zu komplex

Durch die Menge der Aspekte, die in einem Business Process Model kombiniert dargestellt werden, wird das Dokument und seine Erstellung sehr komplex. Es ist extrem schwierig, während der Modellierung einzelner Prozesse gleichzeitig eine stimmige Strukturbeschreibung des Gesamtsystems zu entwickeln. Eine solche Arbeit setzt eine überdurchschnittliche Abstraktionsfähigkeit voraus und ist für viele eine Überforderung.

Die Komplexität des Modells schränkt zusätzlich seinen Nutzen als Übersichtsdokument ein. Es ist für die meisten Beteiligten schwer lesbar. Einzelne Aufgaben können gut verfolgt werden ("was passiert mit einer Mahnung?"). Ein Überblick über das gesamte System kann mit einem Business Process Model nur schwer gewonnen werden.

3.3 "Legacy"

Prozessmodelle werden typischerweise gemeinsam mit den Nutzenden der bestehenden Systeme erstellt. Dabei wird in den meisten Fällen von bestehenden Prozessen ausgegangen und es wird versucht diese zu verbessern. Die bestehenden Prozesse sind entstanden unter den Voraussetzungen der damaligen Umgebung (Organisation, Technologie, Kommunikationskanäle, Medienbrüche etc.). Viele Prozesse sind wie sie sind, weil es *unter den damaligen Umständen* die beste (pragmatischste) Lösung war. Diese Prozesse haben über die Jahre die Arbeitsweise der Leute, die damit arbeiten, geprägt. Viele Mitarbeitende haben diese Prozesse als Grundlage ihrer Arbeit kennen gelernt und verinnerlicht. Diese Prozesse *sind* ihre Arbeit. Aus ihrer Sicht sind die Geschäftsabläufe das, was durch die bestehenden Prozesse vorgegeben wird. Ein gutes Beispiel dafür ist die schweizerische Postleitzahl. Sie wurde 1964 eingeführt um die manuellen Prozesse der Postsortierung

und -verteilung zu verbessern. Ihre (hierarchische) Struktur war auf geografische Regionen, wirtschaftliche Zentren und die Hauptverkehrsachsen (vor allem der Bahn) ausgerichtet. Bei der Einführung von der Bevölkerung bekämpft, wandelte sie sich mit der Zeit zu einem identitätsstiftenden Kulturgut. Bei der heutigen Technik der optischen Adresserkennung und automatischen Sortierung ist sie ein ineffizientes Überbleibsel. Trotzdem wird sie in Prozessen rund um Adressierung weiterhin eingesetzt und verwendet.

Bessere Prozesse entstehen, wenn aus den bestehenden Prozessen die rein fachlichen Ziele (=Informationszustände) extrahiert werden und der Weg, diese zu erreichen unter Einbezug der aktuellen Rahmenbedingungen neu entworfen wird.

3.4 Mehrdeutige Begriffsdefinitionen

Beim Modellieren von Geschäftsprozessen werden Geschäftsbereiche immer aus Sicht einzelner Prozesse und damit einzelner Akteure oder Organisationseinheiten betrachtet und modelliert. Die Begriffe, die verwendet werden um die benötigten oder erzeugten Informationen (Entitäten) zu beschreiben, werden immer aus Sicht der momentanen Gesprächspartner interpretiert. Beispiel: Jeder und jede weiss, was ein “Produkt” ist. In einer Applikation, die den gesamten Produktezyklus abhandelt, ändert der Begriff von Abteilung zu Abteilung seine Bedeutung. Ein “Produkt” ist in der Herstellung etwas anderes (sprich hat andere Eigenschaften und Beziehungen) als ein “Produkt” im Einkauf, im Marketing, in der Logistik oder im Verkauf.

Es braucht eine von den Akteuren und Prozessbeschreibungen unabhängige Definition und Beschreibung aller gemeinsam genutzten Begriffe. Ohne diese Definitionen besteht die grosse Gefahr, dass unsaubere Benennungen und Definitionen übernommen werden. Es entsteht eine semantisch falsche Architektur.

Das zu frühe Modellieren der Prozesse erschwert die notwendige Diskussion um einheitliche Begriffe und Definitionen. Der jeweilige Fokus auf einzelne Abläufe impliziert eine bestimmte Interpretation der Begriffe. Missverständnisse in der Kommunikation zwischen Fach und IT aber auch zwischen verschiedenen Fachbereichen, bleiben dadurch während der Business Analyse häufig unentdeckt.

4 Ursachen und Lösung

4.1 Ursachen

Warum beginnen fast alle Entwickler und Entwicklerinnen in allen Stufen der Softwareentwicklung mit den Prozessen und leiten daraus die benötigen Daten ab? Warum ist diese *falsche Reihenfolge* so beliebt? Wir wissen spätestens seit Niklaus Wirth’s *Algorithmen und Datenstrukturen (1975)* dass eine gute Datenstruktur zu einfacheren Algorithmen führen kann.

Es scheint einfacher zu sein, Prozesse (Wege, Touren) zu beschreiben als statische Strukturen (Karten). Eine überwiegende Mehrheit der Menschen beschreibt lieber Touren (siehe [LL75]). Interessanterweise ist es im Gegensatz dazu einfacher, sich ein Bild einer Umgebung anhand einer Karte zu machen, als anhand der Beschreibung einer Tour durch diese Landschaft zu navigieren. Diese Asymmetrie erklärt sich damit, dass auf einer Karte mehr Information auf kleinerem Raum dargestellt ist und dass sich mit dem Verständnis der zugrunde liegenden Struktur die Touren fast von selbst ergeben. Information verdichtet zu dokumentieren ist aufwändig. Dieser Mehraufwand wird durch Einsparungen beim Lesen wettgemacht. Goethe hat diesen Sachverhalt so beschrieben: “Entschuldige die Länge des Briefes, ich hatte keine Zeit mich kurz zu fassen!”

Einheitliche Definitionen und Begriffsverwendungen sind schwierig zu finden und durchzusetzen. Sie setzen voraus, dass die Differenzen erkannt sind und dass die Beteiligten bereit sind, ihre Begriffsverwendungen und damit ihre Sprechweise anzupassen. Eindeutige Begriffe können nur in Diskussionen über Inhalte und Verständnis erarbeitet werden. Häufig braucht es fachliche Entscheidungen über die richtige bzw. in dieser Umgebung sinnvollste Verwendung eines Begriffs. Diese Diskussionen werden in den Sitzungen mit Fachvertretern gern vermieden. Sie bedeuten viel Arbeit und der Aufwand für die möglichen Konsequenzen wird von Informatikern und Anwendenden gescheut. Es ist vordergründig einfacher (= mit weniger Arbeit verbunden), sich über Prozesse zu unterhalten. Der Kontext und die möglichen Interpretationen sind klarer und es finden weniger Diskussionen statt.

Es ist ein grosser Aufwand, in bestehenden Prozessen die unterschiedlichen Aspekte zu identifizieren, extrahieren und ändern. Es ist eine intellektuelle Herausforderung und braucht gute Fachkenntnisse. Es müssen oft Widerstände der Nutzenden und ihrer Manager überwunden werden (siehe das Beispiel PLZ auf Seite 182). Diese Vermeidung von schwierigen Aufgaben hat zur Folge, dass zu viele überflüssige und veraltete Anteile der bestehenden Geschäftsprozesse in die neue Lösung übernommen werden.

Der Versuch (Denk-)Aufwand zu vermeiden, sowie die Abneigung gegen eine Revision des eigenen Verständnisses der Dinge, führen zu unsauberen Begriffen und Datenstrukturen und damit zu suboptimalen, stark an die Vergangenheit angelehnten (*legacy*) Prozessen.

4.2 Lösungsansatz

Die Gesamtaufgabe des BPM muss unterteilt werden. Häufig werden strukturelle Kriterien (Systemgrenzen, Medienbrüche, Organisationseinheiten, Programmierumgebungen etc.) für die Unterteilung in Teilaufgaben angewendet. Diese haben den Nachteil, dass mit jeder Änderung der Umgebung die Modulstruktur der Lösung ineffizient oder falsch werden kann. Der stabilste Aspekt von Systemen ist die eigentliche Fachsemantik. Die Informationsstrukturen und fachlichen Grundabläufe einer doppelten Buchhaltung z.B. haben sich seit ihrer Einführung im 14. Jahrhundert kaum verändert.

Am volatilsten sind Technologien und Medien. Eine Strukturierung der Lösung nach semantischen Aspekten führt zu langfristig stabilen Systemen. Für eine stabile und einfache Lösung müssen:

- Aspekte mit unterschiedlicher Änderungsrate in gesonderten Arbeitsschritten und Dokumenten dokumentiert werden.
- Prozessvoraussetzungen und Ergebnisse (Ereignisse, Start- und Zielzustände) vor den Prozessen modelliert und beschrieben werden.
- Aspekte, die der Kommunikation mit den Auftraggebern oder Fachnutzern dienen, getrennt werden von Aspekten für technische Belange.
  (Jedes Dokument darf nur Elemente enthalten, die das adressierte Zielpublikum als relevant betrachtet und verstehen kann.)

Das folgende Vorgehen hat sich für BPM bewährt. Als erstes werden Geschäftsereignisse[2] und die möglichen Systemzustände (Informationsmodell) erstellt. Als Voraussetzung für die Modellierung von guten Prozessen können und sollen diese beiden Dokumente unabhängig von Prozessbetrachtungen erarbeitet werden. Als nächster Schritt werden die stabilen, rein fachlichen Aspekte der Geschäftsprozesse beschrieben. Möglichst unabhängig von jeglicher Technologie, Organisation und Medien(-brüchen) etc.

Erst nach Vorliegen dieser Übersichtsergebnisse aus fachlicher Sicht, können gute betriebliche Prozesse erstellt werden, die alle Aspekte kombinieren. Dieser letzte Schritt bettet die fachlichen Anforderungen (Geschäftsereignisse, Informationsstruktur, fachliche Prozessbeschreibung) in die aktuelle Umgebung (Organisation, Technologie, Systemlandschaft) ein.

Abbildung 3 und 4 zeigen Informationsbedarf und Fachprozessschritte des Entwicklungsprozesses in rudimentärer Form.

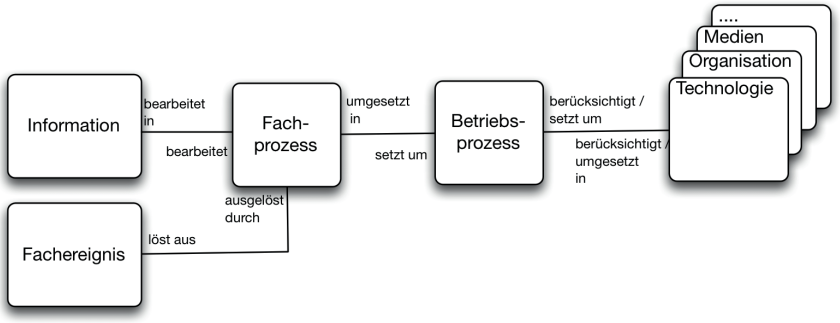


Abb. 3: Skizze eines Informationsmodells des Entwicklungsprozesses

[2] Geschäftsereignisse sind fachliche Umstände die Geschäftsprozesse auslösen. Da diese Ereignislisten eine einfache Auflistung sind, wird ihre Erarbeitung und Dokumentation in diesem Paper aus Platzgründen nicht weiter behandelt.

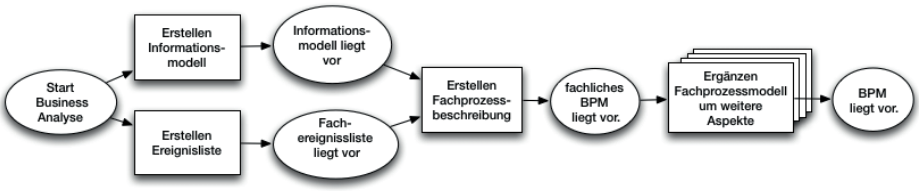


Abb. 4: Skizze eines Fachprozessmodells des Entwicklungsprozesses

5 Zusammenfassung

Nicht BPM per se ist schädlich. Es wird in der Praxis falsch bzw. zum falschen Zeitpunkt eingesetzt. Es werden zu viele Aspekte in der falschen Reihenfolge und mit falschem Fokus bearbeitet. Dies führt dazu, dass

- keine benutzertauglichen Übersichtsdokumente entstehen.
- viele Business Analysten, Manager und Fachleute von ihren Aufgaben überfordert sind.
- die resultierenden Datenmodelle die Fachsemantik nicht optimal abbilden.
- zu viele *legacy*-Aspekte in die neue Lösung übernommen werden.
- viele Module zu komplex werden.
- neue Technologien, Organisationsformen und Medien nicht effizient genutzt werden können.

Der im Abschnitt 4.2 beschriebene Lösungsansatz wurde in vielen Projekten erfolgreich angewendet. Die Modellierung der Prozesse wurde massiv vereinfacht. Viele Prozesse wurden im Vergleich zur alten Lösung einfacher. Die obige Negativliste wurde aus Erfahrungen mehrerer Projekte extrahiert. Meine Hypothese ist, dass ein Wechsel von BPM zu Informationsmodellierung als erster Schritt im Entwicklungsprozess diese Mängel vermindert oder verhindert. Ich hoffe, dass die vorgelegten Überlegungen Leute motivieren, nach diesem Ansatz zu arbeiten bzw. ihre Arbeitsweise und Ergebnisse zu kommunizieren, und dass sich dadurch genügend Beispiele ergeben, damit die Vorteile des Vorgehens objektiver bestätigt werden können.

Literaturverzeichnis

[BP10] BPMN 2.0 by Example. Version 1.0, www.omg.org/spec/BPMN/2.0/examples/PDF/10-06-02.pdf Page 26.

[Di68] Dijkstra, Edsger W.: Go To Statement Considered Harmful. Communications of the ACM Volume 11 Issue 3, S. 147–148, 1968.

[LL75] Linde, Charlotte; Labov, William: Spatial Networks as a Site for the Study of Language and Thought. Language Vol. 51, No. 4, S. 924–939, 1975.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 189

Automatically Binding Variables of Invariants to Violating Elements in an OCL-Aligned XBase-Language

Sebastian Fiss[1], Max E. Kramer[1], Michael Langhammer[1]

Abstract: Constraints that have to hold for all models of a modeling language are often specified as invariants using the Object Constraint Language (OCL). If violations of such invariants shall be documented or resolved in a software system, the exact model elements that violate these conditions have to be computed. OCL validation engines provide, however, only a single context element at which a check for a violated invariant originated.Therefore, the computation of elements that caused an invariant violation is often specified in addition to the invariant declaration with redundant information. These redundancies can make it hard to develop and maintain systems that document or resolve invariant violations.
In this paper, we present an automated approach and tool for declaring and binding parameters of invariants to violating elements based on boolean invariant expressions that are similar to OCL invariants. The tool computes a transformed invariant that returns violating elements for each iterator variable of the invariant expression that matches an explicitly declared invariant parameter. The approach can be used for OCL invariants and all models of languages conforming to the Meta-Object Facility (MOF) standard. We have evaluated our invariant language and transformation tool by transforming 88 invariants of the Unified Modeling Language (UML).

Keywords: OCL, Xbase, XOCL4Inv

1 Introduction

When models are used to develop software systems, a metamodel can be used as a language specification that constrains all valid model instances. Not all types of constraints that have to be enforced can directly be expressed in a metamodel. For such constraints, specification languages can be used in addition to the metamodeling language. If a constraint has to hold for all models of a language, it is usually called an invariant. The Object Constraint Language (OCL) defined in the ISO 19507 standard is a popular language for defining such invariants for object-oriented software . OCL invariants are mainly used to validate model instances in order to ensure correctness prior to further processing or manipulation.

For many application scenarios, it is not sufficient to know whether an invariant holds for a given model: In order to document or resolve the problem indicated by an invariant violation, it is important to obtain the context of a violation [KPP09].In OCL, an invariant is, however, only a boolean constraint expression that may specify a name and a context type. Therefore, OCL-based validation engines provide only a single context element of this type to indicate an invariant violation. Many OCL invariants navigate and inspect several

[1] Karlsruhe Institute of Technology (KIT), Institute for Program Structures and Data Organization (IPD), Chair for Software Design and Quality (SDQ), Am Fasanengarten 5, 76131 Karlsruhe, Germany, sebastian.fiss@student.kit.edu, max.e.kramer@kit.edu, michael.langhammer@kit.edu

different elements and collections of elements related to the context element at which a check was initiated. Therefore, the context element of an OCL invariant often does not directly indicate where, how, and why a model violates the constraints. To achieve this, more specific elements that cause an invariant violation, e.g. by not satisfying one of several conditions defined in the invariant, have to be retrieved.

The retrieval of such model elements that cause an invariant violation is usually defined separately from the boolean invariant condition. As a result, model navigation statements and condition checks are repeated in the code for element retrieval and constraint validation. Although only a few statements may be redundant for a single invariant, the amount of duplicated code can grow to a considerable size for metamodels with hundreds of invariants, such as the Unified Modeling Language (UML) [ISO12a]. This code duplication can be a source for costly errors and can lead to unnecessary development and maintenance effort. It is common to all current approaches except EMF-IncQuery, which only support queries.

In this paper, we present an approach and tool[2]. to avoid this code duplication by computing elements that cause an invariant violation directly from an OCL-aligned invariant definition with explicit parameters. We propose a prototypical invariant language XOCL4Inv, which extends the expression language Xbase [Eff+12] and seamlessly integrates with the popular model transformation language Xtend[3]. As a result, our language is extensible and inherits the power and expressiveness of Xbase, e.g. lambda expressions and extension methods.

The language supports the definition of OCL-aligned constraints using boolean expressions and is in large parts syntactically and semantically equivalent to a subset of OCL. In order to relieve developers from writing separate code for element retrieval, it adds a possibility to define which elements should be retrieved in case of an invariant violation: Variables that are used to iterate over collections can be declared as invariant parameters, which is not possible in OCL. For every declared parameter, our tool computes a transformed invariant, which collects all those elements that are a) bound to the iterator variable corresponding to the parameter and b) causing the invariant violation. It is, however, also possible to directly specify queries, if this is preferred to invariant-to-query transformations.

Consider a simplified example invariant for a library metamodel, which specifies in the context of a reading room that all those books in a reading room that are used as reference copies have to have at least three copies:

```
self.books.forall[Book b | b.referenceCopy implies (b.copies >= 3)]
```

Our tool computes a transformed invariant, which collects all books that violate the constraint, i.e. that are used as reference copies but have less than three copies. It replaces the `forall` iterator with a `select` iterator and negates the condition:

```
self.books.select[Book b | !(b.referenceCopy implies (b.copies >= 3))]
```

For more complex invariants, e.g. with more iterators and parameters, it would be a waste of time to specify such transformed invariants manually as they can be computed automatically.

The presented invariant transformation algorithm can be used for OCL invariants that are defined for models conforming to the Meta-Object Facility (MOF) ISO/IEC 19508:2014(E)

[2] The language and tool are available as open-source software on http://sdqweb.ipd.kit.edu/wiki/XOCL4Inv
[3] http://www.eclipse.org/xtend

standard. Our current invariant language XOCL4Inv and transformation prototype is based on the Eclipse Modeling Framework (EMF) and can transform invariants for metamodels that were built with the Essential MOF variant Ecore. We evaluated the correctness by transforming all 88 invariants of the UML metamodel of the Eclipse IDE[4] that contain collection iterators but do not use statements that cannot be transformed, such as `allInstances`.

The paper is structured as follows: In Section 2, we explain concepts and languages that are fundamental for our approach. In Section 3, we present our OCL-aligned invariant language XOCL4Inv. In Section 4, we explain the invariant transformation algorithm. In Section 5, we present our evaluation of the invariant language and transformation algorithm. In Section 6, we discuss related work and in Section 7 we draw some final conclusions.

2 Foundations

In this section, we explain the technologies and concepts that we use for our approach.

2.1 Model-Driven Software Development (MDSD)

In MDSD [SV06] models and code are used to develop a software system. An important point is that models are at least as important as the source code and not used for documentation purposes only. A common use case is to automatically create source code from the information in the models. The models are often created by domain experts to model a specific domain. To apply these models, Stahl et al. have described three requirements: First, DSLs are necessary to create the models. Second, model-to-code transformations languages are required to process them. Last, specific compilers, generators or transformers are necessary to create executable code from the models.

2.2 Object Constraint Language (OCL)

OCL [ISO12b] is a typed, declarative language that can be used to describe constraints that apply to model instances. OCL was initially developed for UML models, but can be used for arbitrary metamodels. Constraints that are specified with OCL are side-effect free. This means that the queried model instance is not changed by OCL. In MDSD, OCL is used to define constraints and invariants that cannot be easily expressed in a metamodel. Users of OCL can check whether a specific model instance fulfills the constraints that are defined for the metamodel. If one of these constraints is not fulfilled, the model instance is not valid.

2.3 Xtext and Xbase

Xtext [EV06] is a language development framework for creating DSLs. Users have to define the grammar of their DSL. From the grammar definition Xtext creates the lexer, parser and

[4] Eclipse UML metamodel – Rev. 57c76de64a8925e897c2a2ef0a898ea6c153816d – 2014-12-14 http://git.eclipse.org/c/uml2/org.eclipse.uml2.git/tree/plugins/org.eclipse.uml2.uml/model/UML.ecore

```
1 context ReadingRoom
2 invariant AtLeast3ReferenceCopies (Book b)
3 check self.books.forall[Book b | b.referenceCopy implies (b.copies >= 3)]
```

Listing 1: XOCL4Inv invariant definition with a simplified constraint for a library

the semantic analyzers for the new DSL. Xtext itself as well as the DSLs designed with it are integrated in the Eclipse IDE.

Xbase [Eff+12] is an expression language that can be used within any DSL that is created with Xtext. It is a partial programming language with a Java-like syntax. The goal of Xbase is to reduce the necessary effort to implement a DSL. Xbase expressions are similar to Java expressions and the type system is linked to the Java type system. Since it is an expression language, Xbase does not have the concept of statements in contrast to Java.

3 XOCL4Inv: an Xbase Extension for OCL-Aligned Invariants

We present a prototypical language for invariant specifications, which provides a syntax and look-and-feel that is very close to OCL. Unfortunately, OCL cannot directly be used to automatically retrieve elements that cause an invariant violation as described in Section 4 for two reasons: a) a mechanism for indicating which elements shall be retrieved is needed, and b) the language should be restricted to forbid the formulation of invariants for which the demanded elements cannot be retrieved. Both could also be achieved by extending an OCL grammar, editor, and validation engine. We decided, however, to develop a new language that can easily be extended and integrated with other languages for further research and that leverages the functional programming style and tool-support of Xtend. We first present the structure of our language and then we explain the relation to OCL.

3.1 Language Structure

In XOCL4Inv, model constraints can be declared using invariants. Listing 1 shows the running example. Within the invariant declaration, a context type has to be specified that conforms to a type from the constrained model (`ReadingRoom`, line 1). The constraint must hold for all model instances of the provided type. These context elements are bound by the tool as implicit first parameters for each invariant declaration. Furthermore, a unique name is used as an identifier for the invariant (`AtLeast3ReferenceCopies`, line 2).

In addition to the context element, XOCL4Inv allows the declaration of optional invariant parameters (`Book b`, line 2). Each parameter has a unique name and specifies an element type. These parameters are used to indicate which elements of a certain type need to be bound from the invariant upon its violation. XOCL4Inv allows the specification of multiple invariant parameters which get bound to independent sets of constraint-violating elements.

Finally, the language allows the declaration of a constraint (line 3). A constraint is a boolean expression that has to hold for every model instance of the context element type, which

OCL	Xbase	Expression of extension method for XOCL4Inv
`iterate`	`fold`	-
`forAll`	`forall`	-
`forAll(a,b)`	-	`coll.product(coll).forall(predicate)`
`exists`	`exists`	-
`exists(a,b)`	-	`coll.product(coll).exists(predicate)`
`select`	`filter`	-
`reject`	-	`coll.filter[e|!predicate.apply(e)]`
`collect`	`flatten ∘ map`	-
`collectNested`	`map`	-
`isUnique`	-	`coll.groupBy[function.apply(it)].values .forall[it.size == 1]`
`sortedBy`	`sortBy`	-
`any`	`findFirst`	-
`one`	-	`coll.filter(predicate).size == 1`

Table 1: OCL iterators and corresponding Xbase or XOCL4Inv extension methods

is referenced using the keyword `self` within the expression. The constraint is an ordinary expression of the expression language Xbase used in the DSL bench Xtext.It can contain iterators to specify expressions that are iteratively evaluated for a multi-valued property of a metaclass or another collection. For each iterator, an iterator variable can be used to reference the individual element for each evaluation of the iterator expression. If such an iterator variable is explicitly declared as an invariant parameter, then the algorithm described in Section 4 can be used to transform the invariant in order to collect the violating elements.

3.2 OCL Alignment

Invariant declarations in XOCL4Inv are very similar in their structure to OCL. Both languages allow the specification of an invariant name, a context element, and a boolean constraint. Additionally, XOCL4Inv allows the optional specification of invariant parameters to indicate which model elements shall be retrieved for an invariant violation. Since XOCL4Inv constraints are formulated in Xbase, model elements, attributes, references, operations, collection types and primitive types can be used. Most constructs, such as enumerations, null values, and arithmetic and logical expressions exist in Xbase and OCL.

Furthermore, OCL provides methods marked with the prefix *ocl*, for instance `oclAsType` or `oclIsTypeOf`. These methods either rarely occur within invariant constraints, e.g. `oclIsInState`, or have an equivalent Xbase method, for instance type casts or `instanceof`-checks.To provide equivalent functionality for the remaining OCL operations that are commonly used, we defined extension methods which add custom behavior to existing types. Most of these extension methods operate on multi-valued properties. In Table 1, we show which operations for OCL iterators are already available in Xbase and which had to be added to XOCL4Inv. Other common collection operations that do not iterate over collections are shown in Table 2.

OCL	Xbase	Expression of extension method for XOCL4Inv
includes	contains	-
includesAll	containsAll	-
excludes	-	!coll.contains(object)
excludesAll	-	objects.forall[!coll.contains(it)]
isEmpty	empty	-
notEmpty	-	!coll.empty
size	size	-

Table 2: OCL operations without iterator variables and corresponding Xbase or XOCL4Inv methods

For every OCL operation, we either show a corresponding Xbase operation or an expression implemented in an equivalent XOCL4Inv extension method with the same name.

Like in OCL, constraints formulated in XOCL4Inv have to be side-effect free. Xbase in general does not have this restriction, so the language has to be restricted in order to prevent modifications of the model state through constraint checking. Side-effect free methods can be marked with a `@Pure` annotation. Additionally, library methods that cannot be annotated can be added to a user-defined whitelist for pure methods. The tools checks whether constraints only call methods that are marked accordingly or that are whitelisted. The whitelist and static analysis for side-effect free methods should, however, be improved in future work in order to reduce the amount of false positives.

4 Binding Variables of Violating Elements to Parameters

In this section, we explain how invariants formulated in XOCL4Inv are used to compute constraint-violating elements based on invariant parameters. First, we present our algorithm for transforming invariants to queries on a high level and introduce a running example. Then, we explain the individual steps and transformation rules in detail. Finally, we illustrate how the algorithm works by discussing the transformation of the running example.

4.1 Transformation Overview

Invariants in XOCL4Inv contain a boolean constraint for which named invariant parameters may be specified. For each of these parameters, our automated approach finds the corresponding model elements that violate the constraint, and binds these elements to the parameters. More precisely, our algorithm finds the unique multi-valued collection property that is iterated with an iterator variable and that matches the invariant parameter's name and type. For this collection, only those elements that are responsible for the invariant violation are bound. The tool transforms the invariant into a query that collects the constraint-violating elements and binds them to the parameters by executing several transformation steps.

First, the constraint expression is parsed into a custom expression tree. Then, the specified invariant parameters are matched to expression nodes for iterate operations. For every

```
1 context Library
2 invariant AtLeast3OpenReferenceCopies (Book b, List<Edition> editions)
3 check self.books.select[Book b|!b.stack.closed]
4   .map[it.editions.filter[it.referenceCopy]]
5   .forall[List<Edition> editions|editions.reduce[e1,e2|e1.copies + e2.copies] >= 3]
```

Listing 2: Complete example invariant ensuring at least three copies for open reference books

invariant parameter that needs to be bound, transformation rules are applied to the iterator node and its parent nodes in a copy of the expression tree. The resulting transformed expression tree represents the desired query. In a last step, this expression tree is converted back into a query expression, which is used to bind the computed elements to the parameters.

The presented approach has a few limitations: Currently, only invariant parameters that match an iterator variable can be specified. Other attributes and members of model elements can be used to formulate invariants but they cannot be bound to parameters. The effect of variables and members can also be expressed with iterators. Therefore, this is not a limitation of the expressiveness but an inconvenience. Nevertheless, we plan to support invariant parameters that match members or variables similar to the `let`-statement in OCL in future work. Furthermore, the algorithm has to apply transformation rules to the iterator node and all direct and indirect parent nodes. Only operations that correspond to a node for which a transformation rule is defined are therefore allowed after a parameterized iterate expression. Currently, these operations are `not, and, or, select, map, forall`, and `exists`. No such limitations exist for child nodes, i.e. the partial expressions prior to a matched iterator can be arbitrary Xbase expressions.

4.2 Running Example

In Listing 2, we present a complete version of the library invariant, which we already used to motivate our approach and to explain our language. This complete invariant illustrates more transformation rules (see Section 4.6) and applies to the more precise metamodel presented in Figure 1. In contrast to the metamodel used in the simplified invariant, books do not belong to a fixed reading room but to the library. They are stored in a stack which may be open to the public or not. Furthermore, the flag for reference copies and the attribute for number of copies are no longer specified for a book but for a specific edition of a book.

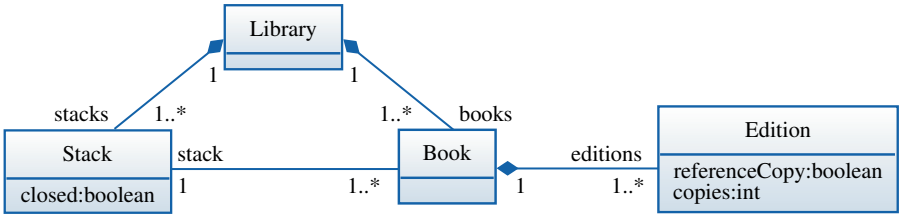


Figure 1: Library metamodel for the complete version of the example invariant

```
1 context Library
2 invariant Books4AtLeast3OpenReferenceCopies (Book b)
3 query self.books.select[Book b|!b.stack.closed &&
4    !(b.editions.filter[it.referenceCopy].reduce[e1,e2|e1.copies + e2.copies] >= 3)]]
```

Listing 3: Query for the complete example returning open reference books with less than three copies

The constraint of the extended example specifies that for every book in an open stack the sum of copies for all editions must total to more than three (line 3–5). If the constraint is violated, the responsible elements have to be computed. A trivial solution would be to return the library context element (line 1). This solution ignores, however, the collection and properties that are inspected during a check and does not determine a precise cause for an invariant violation. The directly responsible elements are those lists of editions for which the sum of copies does not satisfy the constraint. With our approach, these elements could be retrieved by specifying an invariant parameter `List<Edition> editions`. For our running example, we choose the invariant parameter `Book b` (line 2) to obtain those books of the library that have such a list. These indirectly responsible books can be retrieved by a query that is automatically derived from the invariant and shown in Listing 3.

Both example versions illustrate cases in which invariant parameters are needed in addition to contexts. First, there are cases in which an invariant only has to hold for instances with incoming references from the context element: The simplified invariant does not have to hold for books that are not in the reading room. Second, there are cases where a single context as in OCL is not enough because several elements may lead to a violation: Violations of the complete invariant can be resolved by manipulating books or lists of editions.

4.3 Obtaining a Custom Expression Tree Suited for our Transformation

Currently, the XOCL4Inv grammar specifies that invariant constraints can be arbitrary Xbase expressions. Therefore, we obtain an Abstract Syntax Tree (AST) of XExpressions from the parser generated with Xtext. The transformation algorithm defines rules based on much finer distinctions. For instance, all method calls result in a XMemberFeatureCall in Xbase, but method calls have to be transformed in a method-specific way. Calls to the methods `select` or `forall`, for example, have to be transformed differently. Therefore, we use a custom expression tree that differentiates between node types that have to be transformed differently and unifies node types that can be treated identically. This makes it possible to define transformation rules exactly for these node types and to focus on properties that are relevant for the transformation. The metamodel for the nodes of the custom expression tree is shown in Figure 2. A constraint in Xbase syntax is converted into an expression tree that consists of these custom nodes. The nodes' metaclasses are listed in Table 3, along with the XExpression and example expressions from which they are parsed.

The last benefit of a custom tree model are references to parent and child nodes, which are essential for the traversal of the expression tree during the transformation process. These

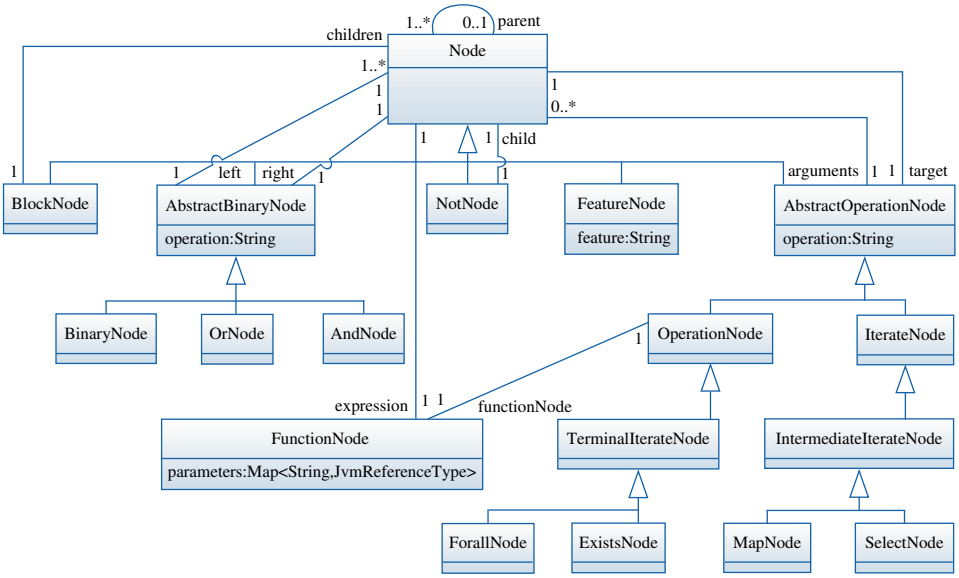


Figure 2: Custom node metamodel for expression trees

references are not contained in XExpressions and make it easier to create, copy, substitute, and modify nodes to apply individual transformation rules.

Currently, the following expressions cannot be transformed because we did not yet define custom node types and transformation rules: type casts, control structures, and variable declarations. Our prototype provides extension methods to transform equivalent constraints that use them instead of the unsupported expressions. In future work, these node types and transformation rules will be added and these extension methods will no longer be needed.

The expression tree for the running example invariant is presented in Figure 3. To obtain the pretty-printed expression shown in Listing 2, an in-order traversal is performed on the tree.

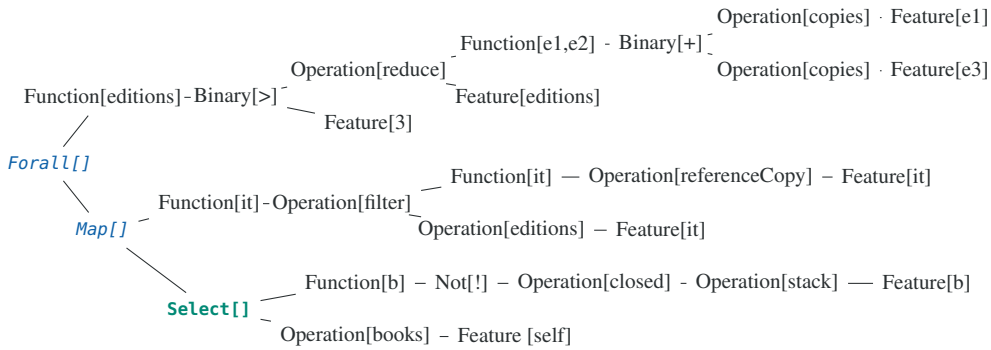


Figure 3: The custom expression tree that is obtained for the complete example invariant

Node type	Example expressions	Corresponding XExpression
ForallNode	`forall`	XMemberFeatureCall
ExistsNode	`exists`	XMemberFeatureCall
SelectNode	`select`	XMemberFeatureCall
MapNode	`map`	XMemberFeatureCall
OperationNode	`self.getBooks(), edition.copies`	XMemberFeatureCall
AndNode	`&&`	XBinaryOperation
OrNode	`||`	XBinaryOperation
BinaryNode	`<, +, /, ...`	XBinaryOperation
NotNode	`!`	XUnaryOperation
FeatureNode	`self, editions, b, it, 3`	XFeatureCall or Literal
FunctionNode	`[a|expression(a)]`	XClosure
BlockNode	`{...}`	XBlockExpression

Table 3: The classification of nodes that are used to build the expression tree

4.4 Matching Parameters to Iterator Nodes

In order to transform the invariant expressions for each specified invariant parameter, the algorithm first matches every parameter to its corresponding iterator node. More precisely, the expression tree is traversed with in-order depth-first search to find all nodes of type IterateNode. If the lambda function of an iterator node specifies an iterator that has the same name as the invariant parameter, the node is a name match candidate. In order to provide only unambiguous matches, both invariant parameter names and iterator variable names have to be unique within the complete invariant constraint.

A name match candidate is only a parameter match if the type of the iterator variable is assignment-compatible to the type of the invariant parameter. This ensures that the resulting query retrieves elements that can be bound to the statically typed invariant parameter. In the running example (Listing 2), the name of the invariant parameter `b` (line 2) matches the iterator variable of the `select` operation (line 3). Both have the same type and therefore the algorithm can proceed. In general, the algorithm finds a matching iterator node for each invariant parameter and transforms a separate invariant copy into a query to retrieve the violations. The required transformation rules are presented in the next section.

4.5 Transforming Iterator Nodes to Queries

Once the expression tree is generated and the matching iterator node is found for a specified invariant parameter, the tool transforms a copy of the expression tree into a tree for a query that selects the desired elements. This transformation is executed independently for every specified unique invariant parameter. The root of this tree is a SelectNode which selects the invariant-violating elements from the invariant context.

Given the constraint expression tree and an iterator node matching an invariant parameter, the algorithm recursively applies transformation rules. It starts top-down at the root node

and transforms child nodes until the iterator node is converted into the desired SelectNode. The query expression is finally obtained by performing an in-order traversal on the resulting query expression tree. The tool uses the transformed expression to bind the elements responsible for a specific constraint violation to the invariant parameter.

In the next paragraph, we explain the individual transformation rules for all transformable node types. The algorithm transforms the parent nodes recursively before transforming the actual node. Therefore, the transformation rules are not isolated but take the transformation result of the parent node into account.

For NotNodes rules of standard predicate logic are applied: Negated conjunctions (AndNode) and disjunctions (OrNode) are transformed by applying DeMorgan's laws. The nodes are replaced with their negated counterparts by pushing the negation inwards. A negated universal quantification (ForallNode) is replaced with an existential quantification (ExistsNode) for the negated predicate, and vice versa.

The ForallNode specifies that all elements in the target collection have to satisfy a given predicate. Therefore, the resulting query selects all elements that do not satisfy the predicate and thus violate the constraint.

```
coll.forall[e | predicate(e)]
-----------------------------
coll.select[e | !predicate(e)]
```

The ExistsNode specifies a predicate that has to be satisfied by at least one element in the target collection. If the constraint is violated, then all elements in the target collection are responsible as none of them satisfies the predicate. But if one element satisfies the predicate, then no elements have to be retrieved even if some of them may not satisfy the predicate.

```
coll.exists[e | predicate(e)]
-----------------------------
coll.select[!coll.exists[e | predicate(e)]]
```

A SelectNode only occurs with a parent node or as the result of a prior transformation. First, the parent node is transformed by applying the appropriate transformation rule to it. The result is a SelectNode for the parent. Then, the predicate of this parent SelectNode is conjunct with the predicate of the current SelectNode and the iterator variables are substituted accordingly to form a single resulting SelectNode.

```
coll.select[e | predicate(e)].select[p | parentPredicate(p)]
------------------------------------------------------------
coll.select[e | predicate(e) && parentPredicate(e)]
```

A MapNode applies a function to each element of the target collection. On this mapped collection, further iterate operations may be used. First, these operations are transformed into a SelectNode. Then, the mapping is inlined into the SelectNode: The MapNode is replaced by the SelectNode and all occurrences of the iterator variable are replaced with an application of the function that was specified in the MapNode.

```
coll.map[e | function(e)].select[p | predicate(p)]
--------------------------------------------------
self.select[e | predicate(function(e))]
```

An AndNode combines an expression that contains the unique iterator variable matching the invariant parameter with another expression. For the resulting query, elements referenced

by this iterator variable have to be retrieved if the expression with the matched variable evaluates to false. Whether the other expression without the matched variable also evaluates to false has no influence on the elements to be retrieved. Therefore, the transformation algorithm removes the expression without the matched variable and only transforms the expression with the matched variable. The order of the expressions does not matter. A swapped invariant `otherExpression && self...e...` is transformed the same way.

```
coll.forall/exists[e | predicate(e)] && otherExpression
-------------------------------------------------------
coll.select[e | predicate(e)]
```

An OrNode combines a parameterized expression and another predicate similar to an AndNode. But in contrast to the transformation for the conjunction, the other predicate of the disjunction cannot be ignored. If the expression evaluates to false but the other predicate holds, then the constraint is not violated. Therefore, the retrieved elements of the child expression may only be selected in the query if the other predicate is violated.

```
coll.forall/exists[e | predicate(e)] || otherPredicate
------------------------------------------------------
coll.select[e | predicate(e) && !otherPredicate]
```

4.6 Transformation Example

To illustrate the transformation we come back to the running example shown in Listing 2 and 3. In order to transform the SelectNode printed in **bold** in Figure 3 with the invariant parameter `Book b`, the algorithm recursively transforms the parent nodes printed in *italics*. It starts at the ForallNode and transforms it into:

```
...select[editions|!(editions.reduce[e1,e2 | e1.copies + e2.copies] >= 3)]
```

The algorithm continues with the transformation of the MapNode. The previously obtained SelectNode is substituted with the following expression:

```
...select[!(it.editions.filter[it.referenceCopy]
 .reduce[e1,e2 | e1.copies + e2.copies] >= 3)]
```

Last, the SelectNode with the invariant parameter is transformed by incorporating the parent node's predicate and substituting the iterator variable:

```
...select[Book b | !b.stack.closed &&!(b.editions.filter[it.referenceCopy]
 .reduce[e1,e2 | e1.copies + e2.copies] >= 3)]
```

The final result is the query presented in Listing 3. It retrieves all books that transitively violate the constraint and is bound to the invariant parameter `Book b` by our tool.

5 Evaluation and Discussion

We evaluated the correctness of the language and of the invariant transformation in two stages: In the first stage, we used synthetic test cases to ensure that invariants formulated in our language produce the same results as the equivalent OCL invariants and that the transformation algorithm produces queries that retrieve the correct elements. In the second stage, we used 88 out of 444 real invariants of the Eclipse metamodel for the UML to check that the language and the transformation algorithm fulfill these properties on them as well.

5.1 Evaluation of the Invariant Language

Our XOCL4Inv language is strongly aligned to OCL, which is commonly used for the specification of metamodel constraints. Therefore, it is necessary that OCL invariants can be expressed in a similar way in XOCL4Inv and produce identical results. We created 19 synthetic language test cases to compare OCL and XOCL4Inv expressions for every operation defined in tables 1 and 2. Each test case provides the same input models to an OCL expression and an XOCL4Inv expression containing Xbase or extension methods and verifies that they retrieve the same results.

In addition to these basic synthetic tests, we also evaluated XOCL4Inv using real invariants from the UML metamodel. It contains 444 constraints, which we categorized individually. 24 constraints have only a textual description, leaving 420 invariants in OCL notation. 175 of these compare attributes or properties of the context element and could be expressed in XOCL4Inv. Further 79 invariants compare sizes of collections from multi-valued properties of the context element. These constraints use the Xbase `size` or `empty` operations. In general, it is not possible to determine the elements that cause an invariant violation for these cardinality constraints because they may be part of the collection or not.

Most importantly, 88 constraints of the UML metamodel contain iterators that can be transformed, i.e. expressions containing `forall`, `exists`, `select`, or `map`. We formulated each of these invariants in XOCL4Inv, as described in the next subsection. A last set of 78 invariants cannot be assigned to any of the previous categories the invariants contain nested combinations of multiple operations or calls to unsupported operations, such as `allInstances`. These nested operations may contain nested invariant parameters, i.e. parameterized iterate operations within the predicate of other iterate operations, which are currently not supported by our prototype. Our approach is able to transform only the 88 invariants that contain non-nested iterate operations that specify an iterator variable. We are currently working on nested expressions by transforming non-nested and nested expressions separately and combining them afterwards.

5.2 Evaluation of the Binding Transformation

In order to test the correctness of the transformation, we checked for 19 additional synthetic transformation test cases that the tool generates the correct query for a given invariant with a parameter and that this query retrieves the correct model elements. For each test, input-output pairs verify that the retrieved elements are equal to the expected ones. The synthetic tests cover all node-operation combinations and the following numbered expression categories:

- 1-2 Unchained parameterized `forall` and `exists` invariants
- 3-4 A `select` with the invariant parameter followed by `forall` (3) or `exists` (4)
- 5-6 A parameterized `map` followed by `forall` or `exists`
- 7-8 Both `&&` and `||` combining a parameterized `forall` with a second predicate
- 9-16 The negation of 1-8
- 17-19 The operations `forall` and `exists` as well as conjunctions and disjunctions in combination with parent nodes

The transformative approach is evaluated on all 88 invariants of the UML metamodel that contain iterators that can be transformed or rewritten to be transformed. For each invariant, we provide an equivalent formulation in XOCL4Inv and manually checked that the generated query equals the expected outcome of the transformation algorithm.

5.3 Discussion

The language and transformation evaluation results for the UML metamodel are promising but have to be complemented by evaluations on further metamodels and further invariants. We demonstrated successfully that we can express and transform 88 invariants of a popular UML metamodel, but it is unclear whether all other Ecore metamodels and OCL invariants can be processed. Therefore, additional invariants should be used to confirm that the language and transformation also work for OCL expressions that are not used in the UML invariants and for structural patterns that do not occur in the UML metamodel.

Furthermore, the transformation evaluation for the UML invariants is based on a manual inspection of the obtained query. In future work this inspection should be automated in order to ensure systematically that the obtained query retrieves the correct elements for several input models. It is, however, an open question whether some of these input models and the sets of elements to be returned for violations can be generated or whether they have to be created manually and can only be automatically compared.

If the transformation is extended in order to transform further expressions, this extension should be evaluated using some of the 78 invariants of the UML metamodel that can currently not be transformed, but also using further invariants of other metamodels. A formal proof of correctness could be done for each transformation rule, but as it mainly realizes well-known predicate logic the benefit of such proofs is disputable.

6 Related Work

Sigma [KC12] is a hybrid model transformation library or internal domain-specific language for Scala. It supports declarative transformation rules and imperative validation and transformation code. Sigma groups constraints in validation contexts and provides facilities to define severity levels for invariant violations, error messages and repair actions. If model elements that caused an invariant validation are used in such repair actions, then the computation of these elements has to be explicitly defined in addition to the definition of the invariant check [cf. KCF14, p.1613, ll.19–22]. Parts of the checking of an invariant can, of course, be factored out and be reused for retrieving model elements. For most invariants this is, however, much more verbose than specifying constraint parameters in our approach.

The Epsilon Validation Language (EVL) [KPP09] of the Epsilon framework is similar to OCL but overcomes several shortcomings of it. Similar to Sigma, it supports the definition of fix procedures for invariants. These fixes are tightly coupled to a constraint, so that it is not possible to write several fixes for a single constraint without repeating it. In contrast to

our approach, parameters that are defined in invariant checks cannot be reused directly in fixes. They have to be defined and computed again in fixes [cf. KPP09, p.215, ll.47–63].

Furthermore, both Sigma and EVL do not separate the definition of invariant checks from fixes. This may be crucial if different fixes are to be defined for different editors, transformations, development projects, or customers while some of the corresponding invariants may be defined for a metamodel regardless of its usage. We are planning to support such a reuse of invariants in the domain-specific language for model consistency [Kra15], which will integrate the invariants language presented in this paper.

EMF-IncQuery [Ber+12; Ujh+15] is a framework for declarative model queries. It performs incremental graph pattern matching based on Rete networks. IncQuery provides a live validation service that can report constraints validations directly after the modification that lead to it. An annotation can be used to turn an ordinary graph pattern into constraints and to define severity levels or error messages for it. Parameters of a constraint pattern can be designated as keys to identify a violation which is a pattern match. These constraint keys are equivalent to the invariant parameters of the presented approach: They also provide elements that lead to a violation based on explicit constraint parameters and do not force developers to repeat parts of the constraint checking logic in order to obtain these elements. The main difference to our approach is, however, the relation to OCL: If elements that cause an invariant violation shall be computed for pre-existing OCL invariants, the transition to our approach based on an OCL-aligned language should require less effort than implementing these invariants again with IncQuery. In projects where such invariants exists and where developers are already familiar with OCL but not with IncQuery, our approach may be more appropriate. To further ease such a use for legacy OCL invariants we are currently working on a automated conversion from OCL to XOCL4Inv. There is a translation from OCL queries to graph patterns that can be queried using EMF-IncQuery [**bergmann2014a**]. With this approach, it would be possible to modify the patterns that result from an OCL invariant in order to obtain wanted elements that violate the invariant. The goal of the translation was, however, better performance. Therefore, a conceptual mapping from the resulting patterns to the initial OCL invariant may not always be straightforward, in contrast to our approach.

7 Conclusion

In this paper, we have presented an OCL-aligned language for invariants with explicit parameters and an algorithm to bind elements causing an invariant violation to these parameters. First, we have explained why an automated computation of such elements from a redundancy-free invariant definition is important to document or resolve invariant violations without unnecessary effort. Then, we have introduced the XOCL4Inv language that combines the style of OCL invariants with the extensibility and power of the expression language Xbase. We have presented an algorithm that obtains queries that compute elements causing an invariant violation by recursively applying transformation rules for invariant expressions. Last, we have discussed how we evaluated the correctness of our language and

transformation algorithm with a prototypical implementation and invariants taken from a popular UML metamodel.

In future work, we are going to evaluate the language and algorithm with invariants of additional case studies. We are also planning to add transformation rules for invariant expressions and parameters that cannot yet be processed. Finally, we are going to reduce the amount of false alarms for side-effects in invariants.

References

[Ber+12] G. Bergmann et al. “Change-driven model transformations”. In: *Software & Systems Modeling* 11.3 (2012), pp. 431–461.

[Eff+12] S. Efftinge et al. “Xbase: Implementing Domain-specific Languages for Java”. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE ’12. ACM, 2012, pp. 112–121.

[EV06] S. Efftinge and M. Völter. “oAW xText: A framework for textual DSLs”. In: *Eclipsecon Summit Europe 2006*. 2006.

[ISO12a] ISO/IEC 19505-2:2012(E). *Information technology – Object Management Group Unified Modeling Language (OMG UML), Superstructure*. International Organization for Standardization, Geneva, Switzerland, 2012, pp. 1–758.

[ISO12b] ISO/IEC 19507:2012(E). *Information technology – Object Management Group Object Constraint Language (OCL)*. International Organization for Standardization, Geneva, Switzerland, 2012, pp. 1–234.

[ISO14] ISO/IEC 19508:2014(E). *Information technology – Object Management Group Meta Object Facility (MOF) Core*. International Organization for Standardization, Geneva, Switzerland, 2014.

[KC12] F. Křikava and P. Collet. “On the Use of an Internal DSL for Enriching EMF Models”. In: *Proceedings of the 12th Workshop on OCL and Textual Modelling*. OCL ’12. ACM, 2012, pp. 25–30.

[KCF14] F. Křikava, P. Collet, and R. B. France. “Manipulating Models Using Internal Domain-specific Languages”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC ’14. ACM, 2014, pp. 1612–1614.

[KPP09] D. S. Kolovos, R. F. Paige, and F. A. Polack. “On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages”. In: *Rigorous Methods for Software Construction and Analysis*. Vol. 5115. LNCS. Springer Berlin Heidelberg, 2009, pp. 204–218.

[Kra15] M. E. Kramer. “A Generative Approach to Change-Driven Consistency in Multi-View Modeling”. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA ’15. 20th International Doctoral Symposium on Components and Architecture (WCOP ’15). ACM, 2015, pp. 129–134.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[Ujh+15] Z. Ujhelyi et al. “EMF-IncQuery: An integrated development environment for live model queries”. In: *Science of Computer Programming* 98, Part 1 (2015), pp. 80–99.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 205

Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool

Martin Gogolla, Frank Hilken[1]

Abstract: Modern systems and their architectures are getting more complex than ever. Development strategies, like model-driven engineering (MDE), help to abstract architectures and provide a promising way to deal with the complexity. Thus, the importance for the underlying models to be correct arises. Today's validation and verification tools should support the developer in generating test cases and provide good concepts for fault detection. In this contribution, we introduce and structure essential use cases for model exploration, validation and verification that help developers find faults in model descriptions. Along with the use cases, we demonstrate the model validator of the USE tool, a modern instance finder for UML and OCL models based on an implementation of relational logic and present the results and findings from the tool.

1 Introduction

Model-driven engineering (MDE) takes a view on system development focusing rather on models than on code. A model catches a system by abstracting its complexity through reduction of information, however preserving properties relative to a given set of concerns. Today, modeling languages, such as the UML (Unified Modeling Language) which comprises the OCL (Object Constraint Language), have found their way into mainstream software development. Models are the central artifacts in MDE because other software elements like code, documentation or tests can be derived from them using model transformations. Common model quality improvement techniques are model validation ("Are we building the right product?") and verification ("Are we building the product right?") [Bo89]. Among the different aspects of a system to be caught, structural aspects represented by class and object diagrams are of central concern.

The tool USE (UML-based Specification Environment) [GBR07] supports the development of UML models enhanced by OCL constraints. USE offers class, object, sequence, statechart, and communication diagrams. It facilitates class and state invariants as well as pre- and postconditions for operations and transitions formulated in OCL. It allows the modeler to validate models and to verify properties by building test scenarios. One USE component that is in charge for this task is the so-called model validator that transforms UML and OCL models as well as validation and verification tasks into the relational logic of Kodkod [TJ07], performs checks on the Kodkod level, and transforms the obtained results back in terms of the UML and OCL model. The modeler works on the UML and OCL level only without a need for expressing details on the relational logic level, i.e., on the Kodkod level.

[1] University of Bremen, Computer Science Department, E-Mail: {gogolla,fhilken}@informatik.uni-bremen.de

The starting point for our current approach is a structural UML model (class diagram) enriched by OCL invariants. With a small, but versatile running example, we discuss various use cases for model validation and verification: model consistency, property reachability, constraint implication, constraint independence, solution interval exploration, and partial solution completion. For example, *model consistency* means that classes and associations considered together with the OCL constraints can be instantiated in form of an object diagram, or *solution interval exploration* means that not only a single instantiation in form of objects and links is examined but all instantiations are taken into account and may be inspected for validation and verification.

The running example here is rather small, but we have already checked that some use cases work for larger models as well [Go15]. Additionally, all presented use cases do not only benefit the USE tool, but also other verification engines [CCR14, Wi08, Br10, QT06] can be used to perform these tasks. Usually, some modifications to the model constraints or additions of such constraints is enough to adopt the tasks. In comparison to the other approaches, however, our current method has the highest coverage of OCL features and offers the most high-level interactions for the use cases.

The rest of the paper is structured as follows. Section 2 gives background information and recaps application areas of UML and OCL models, introduces our running example, shows the basic functionality of USE, and sketches relational logic and Kodkod. The central Sect. 3 discusses the main validation and verification use cases that can be realized, among them how to check model consistency and consequences from stated constraints. Section 4 puts our approach into the context of known work. The paper ends with a conclusion and future work in Sect. 5.

2 Preliminaries

2.1 Application Areas for UML and OCL

UML together with OCL have been succesfully used for system modeling in numerous industrial and academic projects. Here, we refer to only three example projects trying to indicate the wide spectrum of application options. In our own early work [ZG03], we have specified safety properties of a train system in the context of the well-known BART case study (Bay Area Rapid Transit, San Fransisco). In [Al11], central aspects of an industrial video conferencing system developed by Cisco have been studied. In [OM13], UML and OCL are employed for the specification of the UML itself by introducing the so-called UML metamodel in which fundamental well-formedness rules of UML are expressed as OCL constraints.

2.2 Example UML and OCL Model in USE

The screenshot in Fig. 1 shows how the running example employed here is represented in USE. The class diagram in the top right contains one class `Person` having a *first name*, a

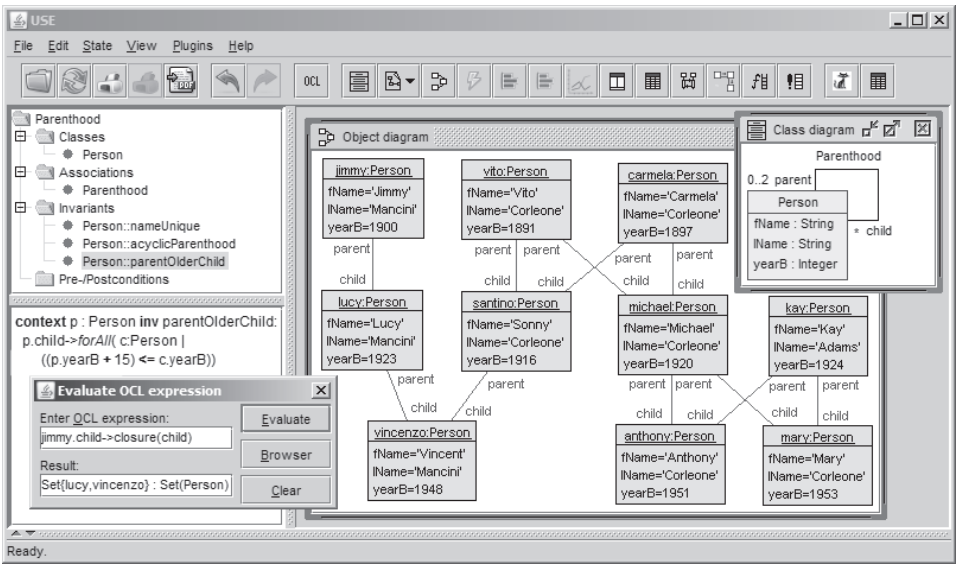


Fig. 1: Parenthood Example class and object diagram.

last name and the *year of birth* as attributes. The association `Parenthood` resembles the parent-child relationship with at most two parents and arbitrary many children per person.

The model may be employed to present family trees as UML object diagrams. To enforce rational family trees three simple invariants are employed, thereby restricting the set of allowed system states, that is defined by the class diagram. These OCL constraints ensure unique names for all persons, prevent cycles in the parenthood relationships and require parents to be at least 15 years older than their children.

```
context p1, p2 : Person inv nameUnique:
  p1<>p2 implies (p1.fName<>p2.fName or p1.lName<>p2.lName)
context p : Person inv acyclicParenthood:
  p.parent->closure(parent)->excludes(p)
context p : Person inv parentOlderChild:
  p.child->forAll(c | p.yearB+15<=c.yearB)
```

The object diagram in the center of Fig. 1, showing several `Person` objects and `Parenthood` links, illustrates a system state of the model showing the family tree as present in the movie *Godfather*. In the lower left, OCL is employed for an ad-hoc query that returns all offsprings of *jimmy*. Arbitrary OCL expressions can be used here to analyze the system state. The present system state satisfies all three invariants and model inherent constraints, such as multiplicity constraints.

2.3 Relational Logic and Kodkod

The verification engine used in this paper, the USE model validator, is based on the relational logic of Kodkod [TJ07]. Kodkod defines a problem to consist of a universe, i.e., a set of

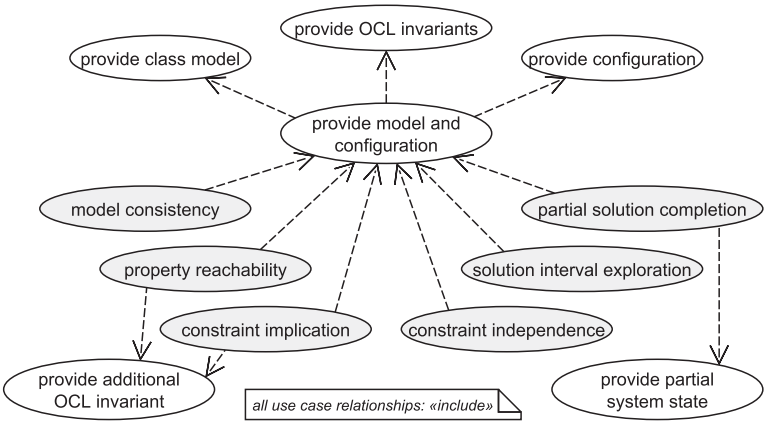


Fig. 2: Validation and verification use cases.

uninterpretable atoms, a set of relation declarations and a relational formula. The universe is defined by the underlying class diagram together with the configuration. The configuration specifies the number of objects available in a solution including primitive data types like `Integer` and `String`. For these types, a minimum and maximum number of instances is defined and specific values for class attributes can be specified. Finally, the relational formula is constructed from (1) UML structural constraints, (2) OCL class invariants and again (3) the configuration. The translation process is based on [KG12].

Given a class diagram and a configuration, the USE model validator generates all three aspects required for Kodkod to solve the problem instance with an off-the-shelf SAT solver. If a valid instantiation is found, the USE model validator generates the corresponding object diagram from the solution instance given by Kodkod. Otherwise, no instance exists for the given model together with the configuration and optionally[1], a counterproof is presented, hinting at possible problems in the model.

3 Validation and Verification Use Cases

The recent validation and verification options in USE will be demonstrated as shown in Fig. 2 by six use cases for central analysis tasks: model consistency, property reachability, constraint implication, constraint independence, solution interval exploration, and partial solution completion.

Figure 3 shows the uses cases from Fig. 2 and the primary input and output artifacts. The distinction between expected and unexpected output parts is as follows: for a use case one typically has in mind a particular expectation for the output of the main flow that is displayed in the upper half of the circle, whereas the output for an alternative flow is pictured in the lower half of the circle. For example, for the *constraint implication* use

[1] Whether a proof is generated depends on the employed SAT solver.

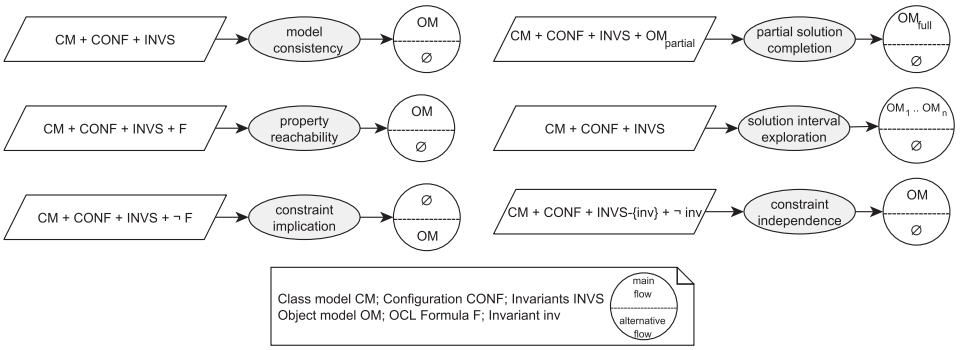


Fig. 3: Use case input and use case output for main and alternative flow.

case the expected output is that no object model is found, whereas a found object model represents a counter example for the suspected constraint consequence.

3.1 Model Consistency

Model consistency is a crucial property. In the context of a class diagram, it guarantees that the UML association multiplicities together with the explicit OCL invariants are not contradictory and that the class diagram can be instantiated with a system state, i.e., an object diagram. Within the context of a UML class diagram, sometimes this property is referred to as class liveness. Finding classes that are not live means that they cannot be instantiated and thus might be unusable in the model.

Model consistency can be proved by handing over to the model validator a configuration that describes the possible populations of classes, associations and attributes in terms of so-called bounds. In technical terms, model consistency is realized through the command `mv -validate <PropertyFile>`[2]. The model validator tries to construct within the specified bounds a valid system state (object diagram). If successful, the system state can be inspected, and if not, the model validator reports that the class diagram cannot be instantiated within the specified bounds. Such an analysis process realizes a verification task for a finite domain. The bitwidth used by the underlying solver for integer arithmetic can be configured in the model validator through the command `mv -config bitwidth:=<NumBits>`.

In the following configuration file used for the consistency use case, exactly 10 persons and 11 parenthood links together with attribute values from the stated enumerations are employed[3]. The object and link numbers and datatype values are exactly as in Fig. 1. The particular datatype values are not bound to particular objects, but the assignment is done by the model validator.

```
Person_min = 10; Person_max = 10;
```

[2] Commands, which are newly introduced, are displayed in a black-on-gray style.

[3] The syntax in the configuration files is partly slightly different. We have decided to use this self-explanatory notation.

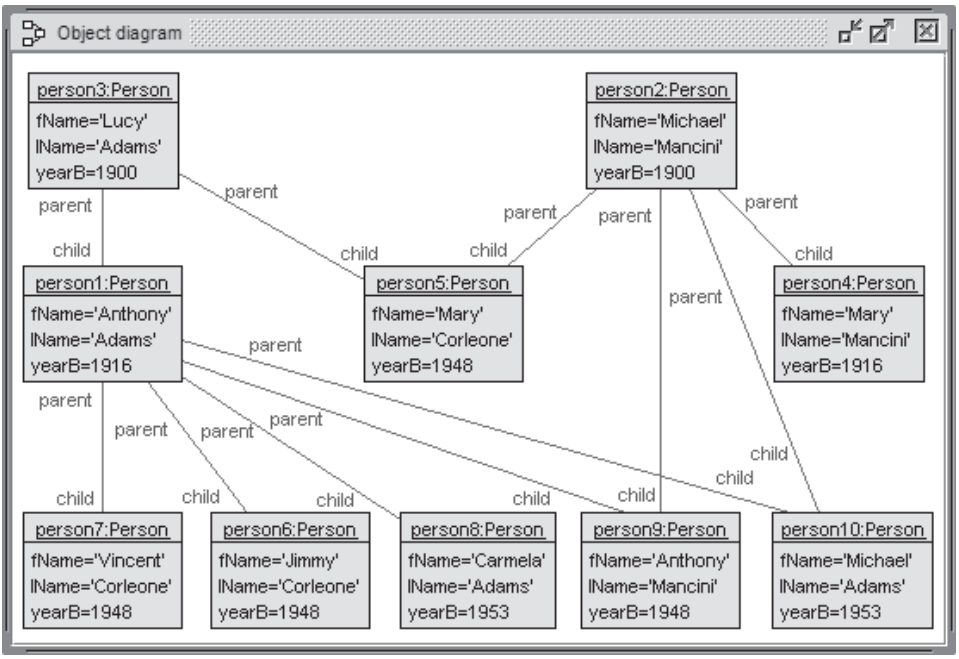


Fig. 4: Generated instantiation for model consistency.

```
Person_fName = Set{'Lucy','Jimmy','Vito','Carmela','Sonny',
    'Michael','Kay','Vincent','Anthony','Mary'};
Person_lName = Set{'Corleone','Adams','Mancini'};
Person_yearB = Set{1891,1897,1900,1916,1920,1924,1923,1948,1951,1953};
Parenthood_min = 11; Parenthood_max = 11;
```

The following protocol shows how USE is fed with the model. The bitwidth configuration in the model validator (`mv -config` command) is necessary due to the desirable realistic year numbers, however it slows down the underlying SAT solver. The validation process is kicked off with the `mv -validate` command, and the constructed object diagram is shown in Fig. 4, which is different from Fig. 1, because the model validator has chosen from the many possible solutions satisfying the specified bounds and datatype values *one* particular solution.

```
use> open parenthood.use
use> mv -config bitwidth:=12
     ModelValidatorConfiguration: Set bitwidth to 12
use> mv -validate corleone.properties
     ModelTransformator: Translation time: 234 ms
     ModelValidator: SATISFIABLE
         Translation time: 359 ms      Solving time: 5351 ms
```

The three time specifications refer to the time needed (a) to translate the class diagram including the invariants into the relational logic of Kodkod, (b) to translate the relational formula and configuration into SAT (this step is performed by Kodkod), and (c) to solve the translated relational formula by the underlying SAT solver. Setting the bitwidth is required

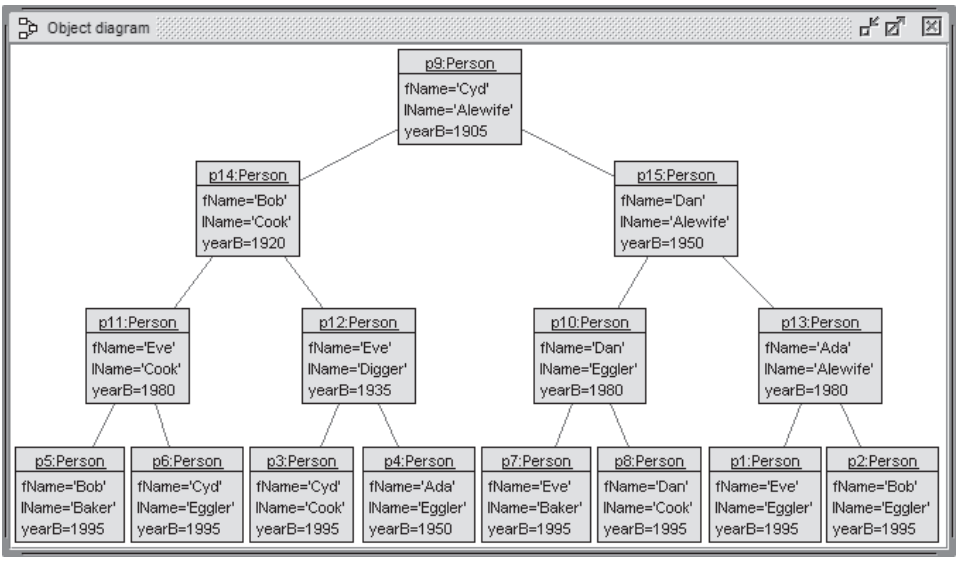


Fig. 5: Generated instantiation for property reachability.

due to the large integers for the years of birth and is necessary in the following scripts, as well. However, we will not repeat all commands below.

3.2 Property Reachability

Property reachability is another verification task that proves that a specific property can be established by object diagrams that are also valid with regard to the original model without having to modify it. Thus the object diagrams of the newly formulated property are a subset of the original object diagrams. The properties are arbitraray OCL expressions that must hold in the generated system state. Additionally, negative properties can be formulated to verify the absence of, e.g., dangerous or illegal system states, or simply unwanted constellations in the system.

In technical terms, property reachability is realized by adding another invariant to the model and by asking the model validator to instantiate the enriched model on the basis of a configuration. `constraints -load <constraintFile>` adds the constraint from the file to the current model. After starting the model validator with the original model enriched by the additional invariant employing the given configuration, the expected result should be a system state that satisfies the original model and the additional specific property.

The following invariant shows that the `Parenthood` model allows object diagrams that constitute perfectly balanced, binary trees. OCL allows to catch the essentials in condensed form.

```
context p:Person inv balancedBinaryTree:
  ---------------------------------------- balance, binary
```

```
  (p.child->size=0 or p.child->size=2) and
  ------------------------------------------------- root
  Person.allInstances->one(r | r.parent->size=0 and
    Person.allInstances->excluding(r)->forAll(p2 |
      p2.parent->size=1)) and
  ------------------------------------------------ balance
  p.child->forAll(c1,c2 | c1.child->closure(child)->size =
    c2.child->closure(child)->size)
```

In the following configuration, exactly 15 person objects are specified, whereas the number of `Parenthood` links is left open. The model validator finds out that exactly 14 `Parenthood` links are needed.

```
Person_min = 15; Person_max = 15;
Person_fName = Set{'Ada', 'Bob', 'Cyd', 'Dan', 'Eve'};
Person_lName =
  Set{'Alewife', 'Baker', 'Cook', 'Digger', 'Eggler'};
Person_yearB = Set{1905, 1920, 1935, 1950, 1965, 1980, 1995};
Parenthood_min = 0; Parenthood_max = *;
```

Specifying exact bounds for `Parenthood` (min 14, max 14) would dramatically speed up the solving process. The following protocol adds the above invariant to the model. The resulting object diagram is shown in Fig. 5. The generated system state confirms the claim, that the property does in fact hold in the running example.

```
use> constraints -load balancedBinaryTree.invs
     Added invariants: Person::balancedBinaryTree
use> mv -validate balancedBinaryTree.properties
     ModelTransformator: Translation time: 296 ms
     ModelValidator: SATISFIABLE
         Translation time: 1576 ms      Solving time: 16396 ms
```

3.3 Constraint Implication

Typically, the modeler specifies a bunch of central properties directly in terms of constraints. However, the modeler often has in mind that the constraint set guarantees that a more global property also holds, i.e., that the global property is an implication of the specified model. In order to formally check the intuitively present global property against the model, the global property is formulated as an invariant, and it is tested whether the suspected implication formally holds.

In technical terms, checking constraint implication is realized by adding the global property to the model. Then that property is logically negated with the command `constraints -flags <Invariant> +n`, and the model validator is asked on the basis of a configuration to instantiate the model. If the global property is an implication from the original model, then the model cannot be instantiated in this situation as the global, implied property has been added in logically negated form. Then the expected result is that the model is unsatisfiable. Otherwise, the model validator will construct a counter example that explains that the suspected invariant is not a model implication.

In the example, the invariant implication which we expect to hold is that a grandparent is at least 30 years older than the grandchild, formulated here as an additional OCL invariant.

```
context gp : Person inv grandparentOlderGrandchild:
  gp.child.child->forAll( gc | gp.yearB+30 <= gc.yearB )
```

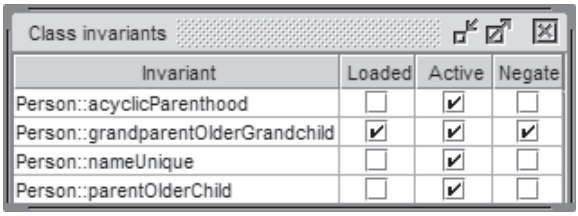


Fig. 6: Status of original and loaded invariants.

The following configuration binds the number of persons to at most 6. The `Parenthood` links are not restricted. Possible attribute values are as above.

```
Person_min = 0; Person_max = 6;
Person_fName = Set{'Ada', ..., 'Eve'};
Person_lName = Set{'Alewife', ..., 'Eggler'};
Person_yearB = Set{1905, ..., 1995};
Parenthood_min = 0; Parenthood_max = *;
```

The protocol to follow adds the previously defined invariant `grandparentOlderGrandchild` to the model and logically negates it. The status of the invariants can be checked either on the command shell or in the USE GUI as shown in Fig. 6. The model validator reports that under the stated configuration the model including the additional negated invariant is unsatisfiable. One could increase the number of possible objects in class `Person` (`Person_max=7, 8, 9, ...`), however this will not change the resulting report. Being convinced that we have performed enough checks, we assume now that the suspected invariant is indeed an implication from the stated model.

```
use> constraints -load grandparentOlderGrandchild.invs
     Added invariants: Person::grandparentOlderGrandchild
use> constraints -flags Person::grandparentOlderGrandchild +n
use> constraints -flags
     -- active class invariants:
     Person::acyclicParenthood
     Person::grandparentOlderGrandchild (negated)
     Person::nameUnique
     Person::parentOlderChild
use> mv -validate grandparentOlderGrandchild.properties
     ModelTransformator: Translation time: 296 ms
     ModelValidator: UNSATISFIABLE
         Translation time: 171 ms      Solving time: 2590 ms
```

3.4 Constraint Independence

Constraint independence is a property of the complete set of constraints. Its goal is to check whether the constraints are independent from each other, i.e., no single constraint is an implication from the other constraints. This property may also be regarded as a kind of minimality property for the constraint set: in this case no single invariant can be removed without changing the set of object diagrams for the class diagram. Independence may or may not hold for the stated constraints. In any case it is interesting to know whether this property holds, for example, in the context of model slicing it will be crucial to reduce the model complexity by identifying a minimal set of needed invariants.

With regard to technical realization, the model validator is started with the option `mv -invIndep <PropertyFile> all`. The result will be a statement for each individual invariant whether it is independent from the other invariants or not. Internally the model validator is started as many times as there are invariants in the model, and in each model validator run exactly one invariant is passed in logically negated form. As a variation of the already discussed `invIndep` option, `mv -invIndep <PropertyFile> <singleInvariant>` (without the keyword `all`) is available in order to construct the example for independence of the single invariant. If an invariant cannot be shown to be independent, further analysis can be performed by deactivating invariants with `constraints -flags <singleInvariant> +d`[4].

Concerning the example, the property file for the independence use case is the same as for the constraint implication use case. Below you see the protocol for calling the model validator with the independence option. You see that the invariants are indeed *not* independent. As detailed in the protocol and shown in Fig. 7, a further analysis with two checks, which deactivate one invariant, reveal that the invariant `parentOlderChild` is implying `acyclicParenthood`.

```
use> mv -invIndep invIndep.properties all
     InvIndepCheck:
       Person::acyclicParenthood: Not Independent
       Person::nameUnique: Independent
       Person::parentOlderChild: Independent
-------------------------------------------
-- nameUnique => acyclicParenthood ?
-- parentOlderChild => acyclicParenthood ?
-------------------------------------------
use> reset
use> constraints -flags Person::acyclicParenthood -d +n
                        Person::nameUnique        -d -n
                        Person::parentOlderChild  +d -n
use> mv -validate invIndep.properties
     ModelValidator: SATISFIABLE
-------------------------------------------
use> reset
use> constraints -flags Person::acyclicParenthood -d +n
                        Person::nameUnique        +d -n
                        Person::parentOlderChild  -d -n
use> mv -validate invIndep.properties
     ModelValidator: UNSATISFIABLE
```

3.5 Solution Interval Exploration

There may be circumstances during validation in which the modeler is not only interested in a single solution in terms of a system state, but the modeler wants to obtain an overview on all solutions. Naturally this will be feasible only if the solution interval is relatively small. By choosing reasonable small bounds for classes and association and by restricting attribute values, interesting results can be achieved: “Even a small scope defines a huge space, and thus often suffices to find subtle bugs.” [Ja06, p. 16].

The technical option for the exploration of a solution interval is accessible in the model validator with the command `mv -scrollingAll <PropertyFile>` in combination with

[4] On the USE command shell, deactivating and negating invariants can be combined.

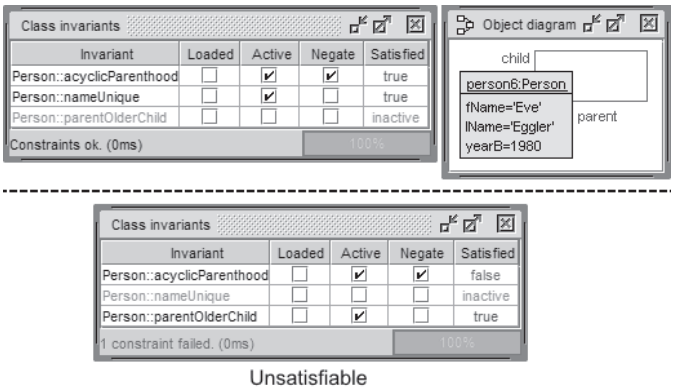


Fig. 7: Invariant status for independence and generated counterexample.

the additional succeeding commands `mv -scrollingAll [prev|next|show(<N>)]`. The first command computes all solutions with regard to the property file. The following commands allow to scroll through the solution interval or to access a solution with the respective solution number (referring to the order in which the solutions have been found).

In the example, the configuration file restricts the number of possible `Person` objects and names to three and the number of age values and parenthood links to two.

```
Person_min = 3; Person_max = 3;
Person_fName = Set{'Ada', 'Bob', 'Cyd'};
Person_lName = Set{'Alewife'};
Person_yearB = Set{1950, 1965};
Parenthood_min = 2; Parenthood_max = 2;
```

The protocol shows that the model validator finds six solutions which are displayed in Fig. 8. These object diagrams represent the complete search space, i.e., all allowed object diagrams of the running example, for the (admittedly and purposely) small configuration.

```
use> mv -scrollingAll scrollingAll.properties
     ModelTransformator: Translation time: 234 ms
     ModelValidator: SATISFIABLE
         Translation time: 1872 ms      Solving time: 187 ms
     ...
     ModelValidator: UNSATISFIABLE
         Translation time: 1622 ms      Solving time: 328 ms
     ModelValidator: Found 6 solutions
use> mv -scrollingAll show(1)   -- show(2) ...
     ModelValidator: Show solution 1
```

We repeat our warning remark with respect to large solution intervals described in the property file when employing the `scrollingAll` option: there may be many solutions; in the example, if the configuration offers one more year (e.g., in total the years `1950`, `1965`, `1980`), then the number of solutions grows from 6 to 36.

If it is too complex to explicitly construct the complete solution interval, one can approximate the interval by computing the next solution in a stepwise manner. The command `mv -scrolling <PropertyFile>` finds a first solution, and following solutions can be obtained by `mv -scrolling next`.

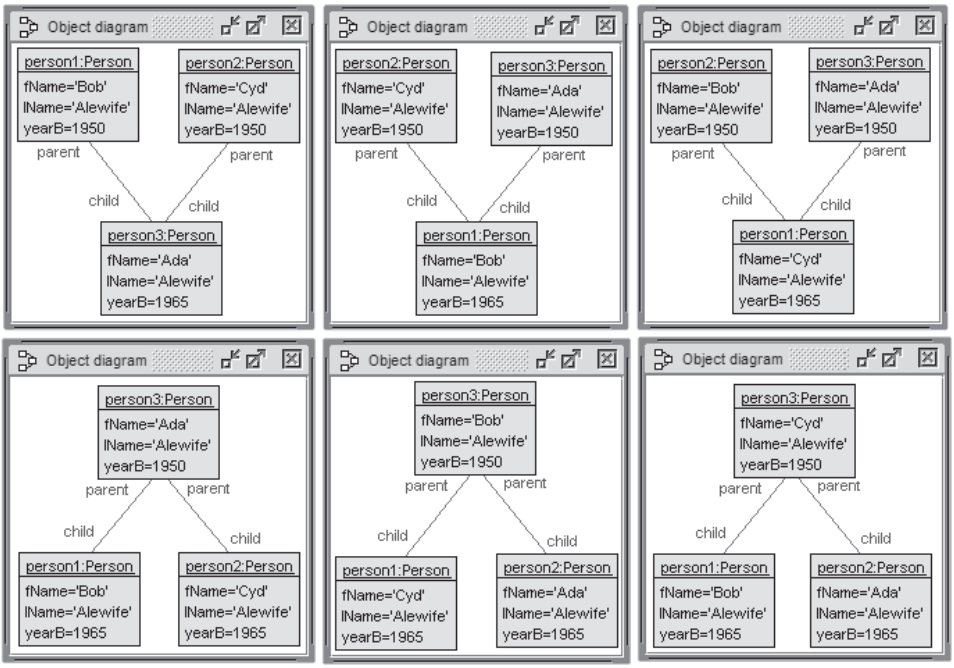


Fig. 8: Solution interval with 6 object diagrams.

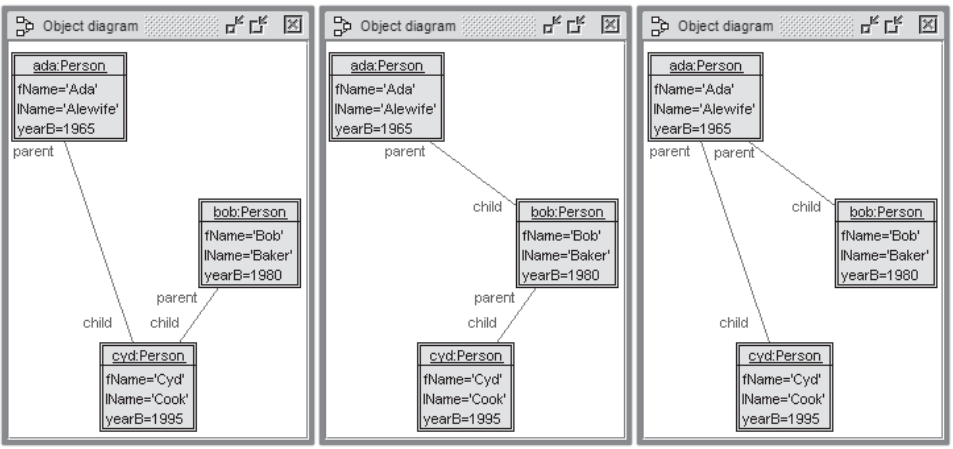


Fig. 9: Completions of partially specified solutions.

3.6 Partial Solution Completion

The last option for a validation and verification task is the completion of a partially specified solution. When one has already constructed objects, attribute values and links (which taken together do not necessarily have to yield a valid system state), one may ask the model validator to complete such a partial system state to a valid solution. If a valid completion with regard to the configuration can be found, a valid system state containing the partially specified system state is constructed. If no valid completion can be found, this is reported to the modeler. An example is shown in Fig. 9.

In terms of the technical realization, the model validator must be explicitly directed to consider the already existing objects and links through specifying `mv -config objExtraction:=on` before the partial system state is asked to be completed. This option may be turned off later, if not needed any more.

The property file fixes the number of `Person` objects to three and the number of `Parenthood` links to two.

```
Person_min = 3; Person_max = 3;
Person_fName = Set{'Ada', 'Bob', 'Cyd', 'Dan', 'Eve'};
Person_lName =
  Set{'Alewife', 'Baker', 'Cook', 'Digger', 'Eggler'};
Person_yearB = Set{1905, 1920, 1935, 1950, 1965, 1980, 1995};
Parenthood_min = 2; Parenthood_max = 2;
```

The protocol file shows the construction of the three objects and fixes their attributes. The links however are not explicitly fixed, but are left as the central construction task for the model validator. The extraction of already existing objects together with their attributes is combined here with the `scrolling` option.

```
use> mv -config objExtraction:=on
     ModelValidatorConfiguration: Enable object extraction
use> !new Person('ada')
use> !set ada.fName := 'Ada'
use> !set ada.lName := 'Alewife'
use> !set ada.yearB := 1965
use> !new Person('bob')
use> ...                          -- bob, cyd analogously
use> mv -scrollingAll completion.properties
     ModelTransformator: Translation time: 202 ms
     ObjectDiagramModelEnricher: Extraction successful
     ModelValidator: SATISFIABLE
         Translation time: 62 ms      Solving time: 16 ms
     ...
     ModelValidator: UNSATISFIABLE
         Translation time: 16 ms      Solving time: 0 ms
     ModelValidator: Found 3 solutions
use> mv -scrollingAll show(1)      -- show(2) ...
```

Figure 9 reveals that three structurally different solutions are found by the model validator. In all three solutions the objects and their attribute values coincide.

Our techniques can also be employed for fault detection. If, for example, a new requirement would be that a person has to have two parents or no parents at all, then adding the constraint `Person.allInstances->exists(parent->size<>2 and parent->size<>0)` and employing the property reachability use case, would lead to

a counterexample disproving the faulty assumption that the new requirement is already granted by the current model.

4 Related Work

The transformation of UML and OCL into formal specifications for validation and verification is a widely considered topic. The approach in [BKS02] presents a translation of UML and OCL into first-order predicate logic to reason about models utilizing theorem provers. Another, similar tool is *UML-RSDS* [LKR10], which allows for the validation of UML class diagrams.

Verification tools use such transformations to reason about models and verify test objectives. *UMLtoCSP* [CCR14] is able to automatically check correctness properties for UML class diagrams enhanced with OCL constraints based on Constraint Logic Programming. The approach operates on a bounded search space similar to the model validator. In [An10], *UML2Alloy* is presented. A transformation of UML and OCL into Alloy [Ja06] is used to be able to automatically test models for consistency with the help of the *Alloy Analizer*. Another approach based on Alloy is presented in [MJOR11]. In particular, limitations of the previous transformation are eliminated by introducing new Alloy constructs to allow for a transformation of more UML features, e.g., multiple inheritance. In [Wi08], OCL expressions are transformed into graph constraints and instance validation is performed by checking models against the graph constraints. Additionally, in [Ca10], a transformation of OCL pre- and postconditions is presented for graph transformations.

The work in [Br10] describes an approach for test generation based on a transformation of UML and OCL into higher-order logic (HOL). With the *HOL-TestGen* tool, test cases(model instances) are generated and validated. In [QT06], a transformation of UML and OCL into first-order logic is described and test methods for models are shown, e.g., *class liveliness* (consistency) and *integrity of invariants* (constraint independence). A different approach is presented in [CGR15]. The authors suggest to use Alloy for the early modeling phase of development due to its better suitability for validation and verification.

Finally, the USE model validator is to a certain degree the successor of the *ASSL* (A Snapshot Specification Language) [GBR05]. ASSL allows the specification of generation procedures for objects and links of each class and association. ASSL searches for a valid system state by iterating through all combinations defined by the procedures. In comparison, the USE model validator translates all model constraints into a SAT formula, which allows for a more efficient generation of a system state, due to detecting bad combinations earlier. Some of the use cases proposed here have been discussed employing ASSL in earlier work [GKH09]. However, the explicit options for formulating the use cases are new, and we employ a new underlying validation engine (Kodkod). In [GBR05] the use case functionalities had to be explicitly formulated in the (programming-like language) ASSL. Now the use cases are basically formulated in terms of (descriptive) configurations.

The approaches mentioned above either already support a subset of the concepts as in the USE model validator or can be used to manually achieve results like *constraint indepen-*

dence or *scrolling*. However, the degree of automation in the current approach is much higher. Without such a high level of automation, validation and verification is a cumbersome task: constraints have to be formulated manually, e.g., in the case of the scrolling use case, one constraint has to be added for every system state found to make sure a different state is generated next. Furthermore, the degree of UML and OCL concept coverage is typically lower in the mentioned approaches.

5 Conclusion and Future Work

In this paper, we have presented techniques to utilize a modern instance finder for a wide range of model validation and verification as well as fault detection methods in UML and OCL models. Examples are shown with the USE model validator using the six use cases: model consistency, property reachability, constraint implication, constraint independence, solution interval exploration, and partial solution completion. The techniques are useful from early development phases to explore models up to testing phases where model properties are verified. For example, the solution interval exploration has proven useful to present example instantiations of a model.

Future work should also concentrate on optimizing the verification tasks by providing help with determining bounds specifically for the presented techniques. Optimizations of the USE model validator itself includes support for more UML features and a more sophisticated handling of strings and large integers. Additionally, not all use cases have a high-level interface for the modeler to use. To make the use cases readily available for everyone, including non-experts, such high-level functions, like `mv -invIndep` for invariant independance, is desirable for all use cases. Finally, larger case studies have to further evaluate the individual methods presented.

References

[Al11] Ali, S.; Iqbal, M. Zohaib Z.; Arcuri, A.; Briand, L.: A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In (Núñez, M.; Hierons, R. M.; Merayo, M. G., eds): Proc. 11th Int. Conf. Quality Software QSIC. IEEE, pp. 41–50, 2011.

[An10] Anastasakis, K.; Bordbar, B.; Georg, G.; Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling, 9(1):69–86, 2010.

[BKS02] Beckert, B.; Keller, U.; Schmitt, P.: Translating the Object Constraint Language into first-order predicate logic. In: Proc. 2nd Verification WS: VERIFY. pp. 2–7, 2002.

[Bo89] Boehm, B.: Software Risk Management. In (Ghezzi, Carlo; McDermid, John A., eds): Proc. 2nd European Software Engineering Conf. (ESEC 1989). Springer, LNCS 387, pp. 1–19, 1989.

[Br10] Brucker, A.; Krieger, M.; Longuet, D.; Wolff, B.: A Specification-Based Test Case Generation Method for UML/OCL. In (Dingel, J.; Solberg, A., eds): Models in Software Engineering. Springer, LNCS 6627, pp. 334–348, 2010.

[Ca10] Cabot, J.; Clarisó, R.; Guerra, E.; de Lara, J.: Synthesis of OCL Pre-conditions for Graph Transformation Rules. In (Tratt, L.; Gogolla, M., eds): Int. Conf. Theory and Practice of Model Transformations. Springer, LNCS 6142, pp. 45–60, 2010.

[CCR14] Cabot, J.; Clarisó, R.; Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software, 93:1–23, 2014.

[CGR15] Cunha, A.; Garis, A. Gabriela; Riesco, D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. SoSyM, 14(1):5–25, 2015.

[GBR05] Gogolla, M.; Bohling, J.; Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Software and System Modeling, 4(4):386–398, 2005.

[GBR07] Gogolla, M.; Büttner, F.; Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program., 69(1-3):27–34, 2007.

[GKH09] Gogolla, M.; Kuhlmann, M.; Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In (Dubois, C., ed.): Tests and Proofs, TAP 2009. Springer, LNCS 5668, pp. 90–104, 2009.

[Go15] Gogolla, M.; Hamann, L.; Hilken, F.; Sedlmeier, M.: Checking UML and OCL Model Consistency: An Experience Report on a Middle-Sized Case Study. In (Blanchette, J. et al., eds.): Tests and Proofs, TAP 2015. Springer, LNCS 9154, pp. 129–136, 2015.

[Ja06] Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press, 2006.

[KG12] Kuhlmann, M.; Gogolla, M.: From UML and OCL to Relational Logic and Back. In (France, R. B.; Kazmeier, J.; Breu, R.; Atkinson, C., eds): Model Driven Engineering Languages and Systems, MODELS 2012. Springer, LNCS 7590, pp. 415–431, 2012.

[LKR10] Lano, K.; Kolahdouz-Rahimi, S.: Specification and Verification of Model Transformations Using UML-RSDS. In (Méry, D.; Merz, S., eds): Integrated Formal Methods, IFM 2010. Springer, LNCS 6396, pp. 199–214, 2010.

[MJOR11] Maoz, S.; J.-O.Ringert; Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In (Whittle, J.; Clark, T.; Kühne, T., eds): Model Driven Engineering Languages and Systems, MODELS 2011. Springer, LNCS 6981, pp. 592–607, 2011.

[OM13] OMG: Unified Modeling Language 2.5. Object Management Group (OMG), http://www.omg.com/uml/, 2013.

[QT06] Queralt, A.; Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. In (Embley, D. W.; Olivé, A.; Ram, S., eds): Conceptual Modeling - ER 2006. Springer, LNCS 4215, pp. 497–512, 2006.

[TJ07] Torlak, E.; Jackson, D.: Kodkod: A Relational Model Finder. In (Grumberg, O.; Huth, M., eds): Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007. Springer, LNCS 4424, pp. 632–647, 2007.

[Wi08] Winkelmann, J.; Taentzer, G.; Ehrig, K.; Küster, J. Malte: Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. ENTCS, 211:159–170, 2008.

[ZG03] Ziemann, P.; Gogolla, M.: Validating OCL Specifications with the USE Tool: An Example Based on the BART Case Study. ENTCS, 80:157–169, 2003.

Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 221

TUnit – Unit Testing For Template-based Code Generators

Carsten Kolassa[1] Markus Look[1], Klaus Müller[1], Alexander Roth[1], Dirk Reiß[2], Bernhard Rumpe[1]

Abstract: Template-based code generator development as part of model-driven development (MDD) demands for strong mechanisms and tools that support developers to improve robustness, i.e., the desired code is generated for the specified inputs. Although different testing methods have been proposed, a method for testing only parts of template-based code generators that can be employed in the early stage of development is lacking. Thus, in this paper we present an approach and an implementation based on JUnit to test template-based code generators. Rather than testing a complete code generator, it facilitates partial testing by supporting the execution of templates with a mocked environment. This eases testing of code generators in early stages of development as well as testing new or changed parts of a code generator. To test the source code generated by the templates under test, different methods are presented including string comparisons, API-based assertions, and abstract syntax tree based assertions.

Keywords: Model-Driven Development; Partial Code Generator Testing; Template-based Code Generation

1 Introduction

With the increasing adoption of model-driven development (MDD) in research and industry [Hu11, Li14], code generation - systematic transformation of compact models to detailed code [FR07] - is gaining importance. To support code generator developers in constructing robust code generators, i.e., code generators that produce the desired code for the specified input, sophisticated mechanisms and tools are required. They have to be integrable into the development process and especially for agile development processes they need to enable partial testing of code generators. Such testing as an essential activity is, however, challenging [St07].

Current approaches for testing code generators (cf. [St07, Jö13, SWC05, Ra10, St06]) require an initial integration effort of the testing procedure, are based on string comparisons only, or are designed to test the code generator as a whole. Other code generator testing approaches employ formal methods [BKS04]. Setting up such tests is time-consuming and once set up they are hard to maintain in an evolving environment, because small changes in the code generator may lead to larger changes in the tests. Consequently, existing testing approaches for code generators are not so easy to use in an agile development environment, where either the code generator does not yet generate complete code artifacts that the tests can validate or only the increment in functionality is to be tested. In summary, exisiting work lacks approaches for partial code generator testing.

[1] RWTH Aachen University, Software Engineering, Germany, http://www.se-rwth.de
[2] TU Braunschweig, Institute for Building Services and Energy Design, Germany

Focusing on template-based code generation, the goal of this paper is to present an approach for testing individual components of template-based code generators that can be employed in early development stage of code generators. We introduce TUnit, an extension of JUnit [JU15] based on the MontiCore [KRV10, Gr08, KRV08] language workbench to support unit testing of code generator templates. In our case, testing a code generator or parts of it means to answer the following questions: *Is the set of specified inputs accepted by the code generator template, e.g., code is generated? Does the code generator template produce syntactically valid source code? Are the target language context conditions valid for the generated source code?* Executing a TUnit test case will run the template under test with a mocked context (e.g. mocked variables, mocked templates, or mocked helper functionality) on (parts of) an input model. This approach allows for testing the output of a single template under test that is part of the overall output of the code generator, rather than testing the whole output of a code generator run. To validate that the template output meets the testers expectations, TUnit provides different kinds of assertion mechanisms including abstract syntax based comparisons and abstract syntax API-based assertions. Additionally, because string comparisons are widely used and sometimes practical, TUnit provides support for such comparisons as well. However, this approach is not robust, as the template output can change on a regular basis, e.g., due to new or deleted whitespaces.

The contributions of this paper are: (a) an understanding of a template engine context (b) concepts for mocking a template's context with nested templates to allow for partial code generator testing in early stage of the development cycle, (c) concepts for abstract syntax based testing of the partial generated source code, and (d) an implementation of these concepts within a widely used testing framework.

The paper is structured as follows: at first, we present an overview of related work (Section 2) and point out their shortcomings. Next, we introduce MontiCore (Section 3), a framework for language processing and code generation, that has been used to implement parts of TUnit. By starting with a basic TUnit test, we point out how template-based code generators can be unit-tested and which challenges need to be solved (Section 4). These challenges are addressed in Section 5. Finally, we conclude our paper in Section 6.

2 Related Work

With the emerging importance of MDD, code generation has received growing attention. In order to support code generator development and ensure code generator robustness, different code generator testing approaches have been proposed and are presented in more detail in [SWC05]. In the remainder of this section, we point out the main ideas of the different testing approaches that target testing of complete code generators.

CoGenTe is a tool for testing code generators [Ra10]. It takes a syntactic and a semantic meta-model of the input language and a test specification, which is a coverage criterion over the meta-model. A generator creates a test-suite that can test any code generator for the particular input language. The generated test-suite is derived using a constraint generator, an inference tree generator, and a constraint solver. Each test-suite comprises

several input models and expected outputs in the target language. To test a code generator, the test-suite input models are passed to the code generator and the generated output is compared to the expected output of the test-suite. In contrast to this approach, we present an approach to test parts of a code generator for predefined input models.

Another approach to test code generators has been proposed in [St06, St07]. It is based on a formal specification of the code generator transformation as a graph rewriting rule and comprises three steps. In the first step - model-in-the-loop - the test model is transformed into an executable model that is simulated. In the second step - software-in-the-loop - the generated model is transformed by the code generator into executable code. Both, the execution results of the simulated model and the execution results of the executed code are finally compared. Existing approaches can be applied to extend this approach by automatically generating the input test-cases [Ze06, Sa08]. In contrast to this technique, our approach uses an instance of the input languge rather than a formal specification. Furthermore, no intermediate model is used for simulation. Our proposed approach works directly on the input model and not only strings but also abstract syntax trees (ASTs) can be compared. Furthermore, an AST-based API is provided that allows to check the generated output.

An instance of the above code generator testing approach to generate JUnit tests has been proposed in [Jö13]. Code generators are modeled as services from atomic service independent building blocks (SIBs). Such SIBs are used to model test cases, which are part of test suites. A code generator transforms the test cases into JUnit test scripts. The execution footprint - basically a string of the SIBs that have been executed - of direct execution of the test data and the execution footprint of the generated and compiled code are compared. The test is successful if the footprints are equal. In this paper, we focus on partial testing of code generators and use, e.g., AST comparisons for validating the generated output.

3 Language Processing and Code Generation with MontiCore

The MontiCore framework [KRV10, Gr08, KRV08] is the foundation for all aspects of language definition, language processing, and template-based code generation in TUnit. In the remainder of this paper, we regard a model as an instance of a language that is processed by the MontiCore framework and used for code generation. The basic structure of the MontiCore framework is shown in Figure 1. The components depicted in the upper left corner including `Grammar`, `MontiCore`, `Symboltable Entries`, `Model`, `Parser+Infrastructure` and `AST` are used for language definition and language processing, i.e., processing an input model. All other components and the components `Symboltable Entries` and `AST` are used for code generation.

The MontiCore framework uses a grammar defining the language to be processed and generates a parser and infrastructure for language processing, which are used to parse models. Each input model needs to conform to the grammar. When reading and processing models, the parser creates an AST that represents their internal structure. This abstract representation of the input model is used for both: further language processing steps and code

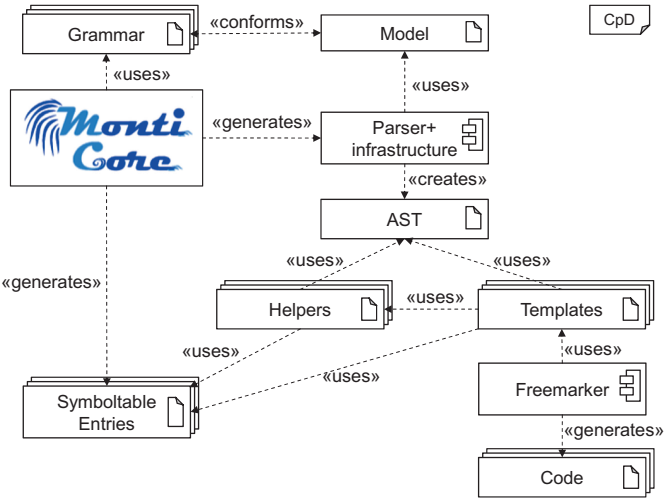


Fig. 1: Overview of MontiCore components for language processing and code generation [Sc12].

generation. Besides the AST, MontiCore uses the `Symboltable Entries` component to create symbol table entries for each symbol of the processed models. Each symbol table entry contains information about the model structure, an element's name, and context information. This stored information is used for referencing symbols in different models and can be used to extend the language processing by defining constraints for the input model or for code generation to retrieve additional information on model symbols.

3.1 Template-based Code Generation with MontiCore

The MontiCore code generation process is based on a template mechanism. Templates written in FreeMarker [Fr15] describe what is to be generated. These templates, which are hierarchically structured via sub-templates, contain target code and FreeMarker expressions that finally produce target code. An overview of a template and its context is depicted in Figure 2. The result of the code generation process is the actual output labeled generated code in the figure.

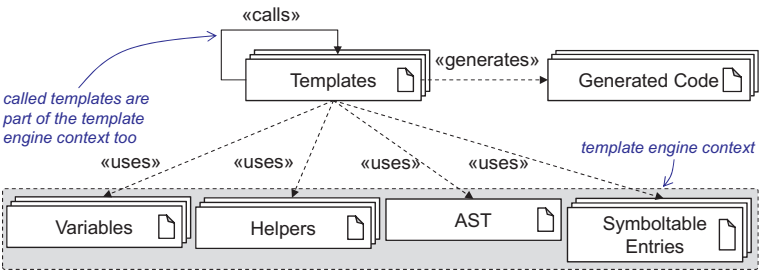


Fig. 2: Elements of the template engine context for template-based code generation in MontiCore.

Each code generation process is started by invoking a root template, which usually calls one or more sub-templates. A template has access to its template engine context object, which contains variables, helper objects, called templates, and symbol table entries, as depicted in Figure 2. A simplified example of a FreeMarker template is given in List. 1. For a better presentation of the template engine context, this listing shows two variables, the ast, and one helper.

FreeMarker

```
1 //Variables: paramType, paramName
2 //Helpers: methodHelper
3 public ${ast.returnType} ${ast.name}
4   (${paramType} ${paramName})
5   ${methodHelper.printThrowsDecl(ast)}
```

List. 1: The (simplified) template for generating a Java method.

The primary input for templates is the AST which is constructed by processing a model file. For presentational reasons, we primarily focus on class diagrams as an input model, i.e., the AST describes the abstract syntax of a class diagram and AST elements are elements from class diagrams including classes, associations, methods, interfaces, and enumerations [Sc12]. A template is called with an AST element of the corresponding model and, in our case, generates Java source code. This AST element can be accessed through the context variable `ast` as shown in an excerpt of a template in List. 1. In this listing, the method `name` of the corresponding AST class is invoked in line 3 to return the name of the model element which is represented by the AST element. This is denoted by the FreeMarker specific syntax `${...}`. Additionally, the template excerpt in List. 1 shows how variables (`paramName` and `paramType`) and helpers (`methodHelper`) are used. The meaning of variables and helpers is explained in more detail in the following.

A template may define local variables, which can be used and modified inside the template. The value for each variable is set when the template is called. For example, the template outlined in List. 1 expects that the values for the variables `paramName` and `paramType` are set when the template is called. The values of these variables are accessed in line 4 to introduce the name and the type of the method parameter into the generated code.

According to the principle of separation of concerns, templates contain target code and simple computations including string concatenations, loops and if-else conditions. In addition to that, further functionality can be implemented in helper classes in Java which are invoked from templates. When a template is called, an instance of the helper class is passed to the template and can be accessed through a helper variable. In List. 1, `methodHelper` in line 5 is a helper variable and it is used to invoke the helper method `printThrowsDecl` which returns the Java throws declaration of the method. The AST is a special kind of helper variable, as it can be used to invoke specific methods from the AST classes.

In order to test templates in isolation, we need to be able to replace either all or only some of the variables and helpers in a template's context with mocked ones. For instance, it might be desirable to apply a specific mock helper class instead of the helper class which would be used by default. Or it might be desired to set the variables to specific values.

4 Code Generator Template Testing with TUnit

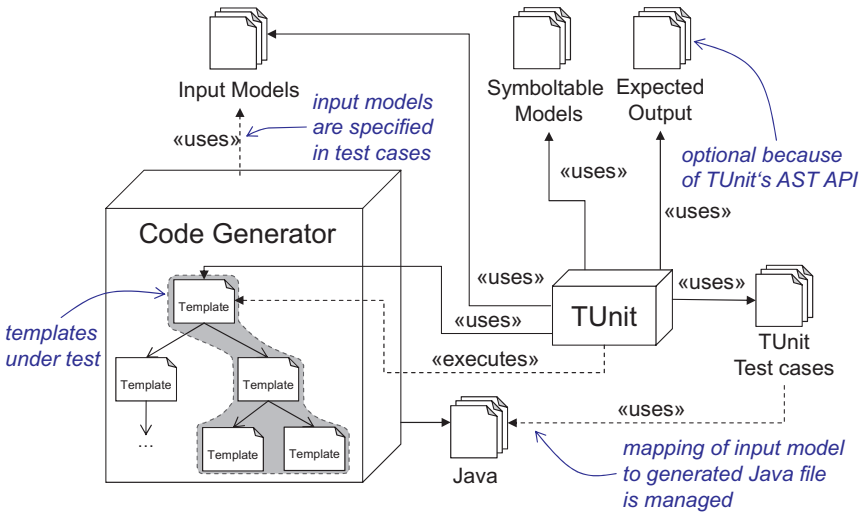


Fig. 3: Overview of our code generator testing approach.

A template-based code generator typically comprises multiple code generator templates in order to generate source code from an input model. For ease of presentation we assume that a code generator accepts one class diagram as input and generates Java source code. In addition, each code generator has access to a symbol table, where the symbols of all symboltable models are stored to identify referenced symbols. Figure 3 depicts an overview of our white-box approach to test code generator templates. In our approach TUnit test cases define the tests, symboltable models used in tests, and the templates under test. Using this input, the templates under test are executed but only for the elements of the model that have been predefined in the TUnit test case. The generated output may then either be compared to the expected output or the TUnit's AST API can be used to define assertions.

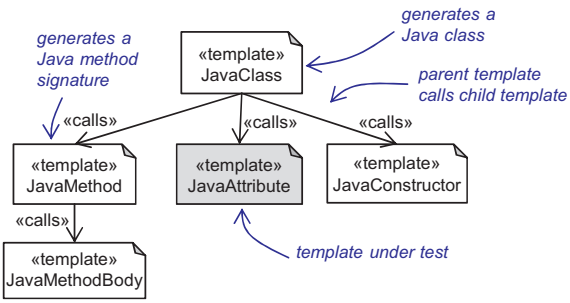


Fig. 4: An example of a code generator's template hierarchy to generate a Java class.

To give a more detailed understanding of how our testing approach works, we assume that the code generator's templates are structured as depicted in Figure 4 and that we want to test the `JavaAttribute` template from the template hierarchy. Its FreeMarker source code is listed in List. 2. For each class diagram attribute that is passed as input to this template

the template generates a Java variable declaration with `public` visibility. For instance, by passing the AST of the class diagram attribute “int attributeName = 5;” the template generates the Java variable declaration statement “public int attributeName = 5;”. Here, the value 5 is the default value. Line 1 of List. 2 generates the variable declaration and line 2 generates the variable instantiation by checking if a value has been defined in the input model. Finally, a semicolon is used to close the Java variable declaration.

FreeMarker

```
1 public ${ast.printType()} ${ast.name}
2   <#if ast.value??> = ${ast.printValue()} </#if>
3 ;
```

List. 2: The (simplified) template for generating a Java attribute.

4.1 Unit Testing Templates

To present the testing concepts for template-based code generators, we extended the JUnit testing framework to support the different testing approaches for code generator templates. Subsequently, we introduce the resulting TUnit and the realized concepts for early stage unit testing templates.

Java

```
1 @RunWith(de.se.rwth.tunit.TUnitRunner.class)
2 @TemplateUnderTest(templateName="JavaAttribute",
3   type = ASTCDAttribute.class)
4 public class TUnitTestClass {
5   @Test
6   @InputModel(fileName = "src/test/" +
7     "resources/input/JavaAttribute.cd")
8   public void templateTestMethod() {  ...  }
9 }
```

List. 3: Skeleton of a TUnit test class with one test method.

In JUnit, test runners are used to execute the test methods implemented in a test class. As TUnit introduces custom annotations that are used for configuration purposes, the default JUnit runner is not appropriate to execute the template tests properly. Due to this, TUnit integrates its own test runner that is aware of the semantics of the annotations and knows how to execute the template tests. Thus, each TUnit test class has to be annotated with the TUnit specific test runner as shown in line 1 in List. 3. This listing shows a complete skeleton for a simple TUnit test class.

Three crucial aspects that are relevant when testing template-based generators are: *Which template is under test? Which input model is used for the template under test? Which parts of the input model are relevant for the test?* TUnit introduces two annotations that define the template under test and the test input model.

The mandatory annotation `@TemplateUnderTest` is used to define the template under test and is used for all test methods of the test class. The annotation provides two mandatory attributes: `templateName` and `type`. In `templateName`, the path to the template under test and its name have to be defined. In `type`, the type of the AST node that is handled by the template has to be stated so that the template is only invoked for the specified type. When executing a template test, each test method is executed by first parsing the specified input model.The created AST is then traversed and the template under test is executed for each AST element that is of the specified `type`. Finally, the test method is actually called. List. 3 shows the TUnit test for the `JavaAttribute` template. This template is invoked for AST nodes of type `ASTCDAttribute`.

The input model – defined with the mandatory `@InputModel` annotation – can either be defined on test class level, then the given input model is used in all test methods, or on test method level, then the input model is only used for that particular test method. In each test method, the template under test will be applied for the corresponding input model.

4.2 Referencing Generated Output by Model Elements

Defining which template and model elements are under test is the first step to test templates. A further essential step in testing templates is to validate that the template output meets the testers expectations. In JUnit, such expectations are expressed using assert methods. For instance, the assert method `assertEquals` ensures that two values are equal or the assert method `assertNotNull` ensures that a specific value is not null.

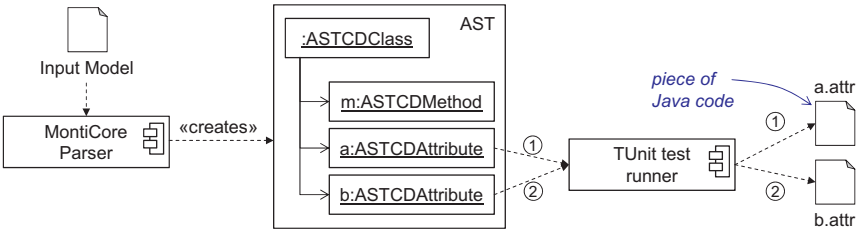


Fig. 5: For each element that fits the specified type of the TUnit test, the template is called and the output is stored in a file.

A prerequisite for being able to formulate such assert statements is that the tester can access the output produced by the template. As explained in Section 4.1, the input model is first parsed when executing a test method. The resulting AST is then traversed and the template is executed for each AST element of the specified type. The output of each template application is stored individually in a distinct file. As the input model for the template may contain multiple elements of the specified type, it is possible that multiple output files are created when executing one test method. Figure 5 shows how TUnit handles the output generated by a template. In this figure, it is assumed that the input model is a class diagram consisting of methods and attributes and that the template under test is defined for AST nodes of type `ASTCDAttribute`. According to Figure 5, the input model contains a class with two attributes. As a result, TUnit creates one file which contains

the output of the template application to the first attribute (*a.attr*) and another file for the output of the template application to the second attribute (*b.attr*).

In a concrete test case, a tester usually wants to validate the expectations concerning a specific output, e.g. the output produced for the first attribute in Figure 5. To accomplish this, one option for a tester would be to construct the name of the output file by himself. This is possible as TUnit creates the output files according to a specific naming convention. However, a disadvantage is that it becomes more laborious to define tests. Moreover, the names of the output files will change as soon as the input model will be updated. Testers would need to update the statically referenced output files after each input model update.

To cope with this problem, TUnit (a) traces which output file was created for which AST element and (b) provides an API that allows to retrieve a particular AST element and that returns the corresponding generated file. In Figure 5, the traceability is depicted by the numbers. Thus, the testers can use the API to uniquely identify a specific AST element and the generated file is returned without expecting the testers to construct the concrete path to the output file on their own. Currently, this API is restricted to class diagram input models. Additionally, the API can only be used to address single model elements only, i.e., a model element can be specified in a fully qualified way.

In order to create a test case for a generator template, we subsequently present two assertion variants that both rely on defining the complete expected output.

4.3 Assertions for Code Generator Templates

The most basic approach is to perform a simple string comparison between the actual output and an expected string. The tester has to define the complete string that is expected as a result of the template application. A disadvantage of this approach is that the testers are forced to denote the complete expected string, which can be quite laborious and error-prone. Moreover, this approach is rather fragile, as every two varying characters will result in a failing test, e.g., whitespace issues. To cope with the latter problem, TUnit offers a more flexible string comparison method which can be configured to neglect specific types of differences, e.g. differences concerning tabulator characters or indentation.

A more advanced method of creating assertions is to perform an AST comparison. In the course of this comparison, it can be ensured that two AST nodes are (not) equal by including not only the AST node itself but also children elements of the AST node. For this purpose the tester has to define the expected output, which needs to be parsed to build the corresponding AST. Moreover, the template output needs to be parsed to build the AST as well. The AST comparison can then be performed based on these two ASTs. It has to be taken into account that the template output can contain only parts of complete files, e.g. a variable declaration. Due to this, a prerequisite for this approach is a parser for the target language and target language constructs.

Comparing two ASTs means to traverse both ASTs and compare the contained objects. Figure 6 illustrates the comparison of a generated and an expected AST. The one on the

left-hand side has been generated by the FreeMarker template shown in List. 2 for the class attribute `int attributeName = 5;`. The AST on the right-hand side of Figure 6 is the AST which was built out of the expected template output. An AST comparison of the generated and the expected AST will reveal the unmatching parts. In Figure 6, this is the variable name and the variable type. As a result, the TUnit assertion will report an error indicating this. A side effect of an AST-based comparison is that the AST of the target language is at hand. This AST can be used to check target language context conditions that may check, e.g. if a variable has been defined before usage. In this way a primary step towards semantically checking the generated code is performed.

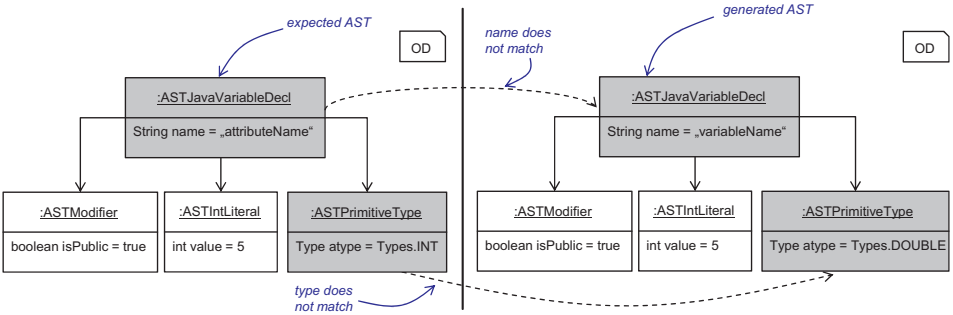


Fig. 6: A comparison of the generated AST (left) and the expected AST (right).

By explicitly stating which template is under test, which input models are used for testing and how the assertions should be handled, template unit testing can be enabled up to the point when the template's context is of relevance for the test. In the following section, the challenge of testing templates that rely on context information is addressed.

5 Context-Aware Unit Testing Code Generator Templates

A code generator template that is under test is not always fully self-contained and thus independent of the template engine context. In other words, it requires certain inputs or values to be accessible during execution. For MontiCore code generators such a context may contain variables, helpers, symbol table entries, and template references. Figure 7 shows the same template hierarchy as Figure 4 but the template under test changed to the `JavaMethod` template, which needs extra context information.

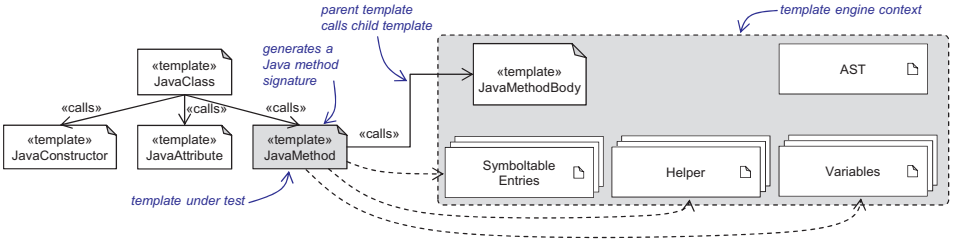


Fig. 7: Testing a code generator template with context.

FreeMarker

```
1 //Variables: paramType, paramName
2 //Helpers: methodHelper
3 public ${ast.printReturnType()} ${ast.printName()}
4  (${paramType} ${paramName})
5 ${methodHelper.printThrowsDecl(ast)}
6 ${tc.include("cd2data.core.templates.JavaMethodBody", ast)}
```

List. 4: The (simplified) template for generating a Java method.

Assuming that the template under test is the `JavaMethod` template, List. 4 shows the FreeMarker source code, which is an extended version of the template excerpt shown before in List. 1. For example, to generate a Java method the template `JavaMethod` is executed with the input "`void methodName(String param){};`", which is stated in the class diagram. The variable `ast` is used to access the elements of the method declaration - in this case the return type of the method and the method name. The parameter type and parameter name are passed to the template as variables. Additionally, the helper `methodHelper` is used to print Java throws declarations. An instance of this helper is passed as well to the template. In addition, a sub template (see line 6 in List. 4) is called to print the body of the method.

While variables, helper, and symbol table entries of a template under test can easily be mocked to provide enough context for the template to be executed in a test, mocking template references influences the depth of the test with respect to the template hierarchy, i.e., the more templates are mocked, the less templates of the overall template hierarchy are tested. For instance, the `JavaMethod` template, which is currently under test, references the `JavaMethodBody` template, i.e., this sub-template is called and its generated code is embedded in the generated code of the parent template. We refer to the mocking of sub-templates as pruning the sub-templates of the template under test.

5.1 Mocking Helpers and Template Variables

In order to mock calls to helper methods, TUnit provides the annotation `@InitHelpers`. This annotation can be used to annotate at most one method in the test class and TUnit expects this method to return a map of strings as keys and objects as values. The strings denote the names of the helper variables and the objects the associated instances of the helper classes. Thus, the tester can define the object to be used when accessing a particular helper variable. He can also implement mocks for helper classes and assign mock objects to the helper variables.

A template can rely on multiple variables that need to be set when calling that template. TUnit supports mocking of variables by providing the annotation `@InitVariables`. At most one method in the test class can be annotated with `@InitVariables` and this method must return a map of strings and strings. The keys of this map denote the variable names to be mocked. The associated values will be used as the variable value when calling the template. In this way, the tester can easily define values for variables needed by a template.

5.2 Mocking Symbol Table

As the symbol table stores information about referenced symbols and is part of the code generator template context, it needs to be mocked for testing as well. For mocking symbol tables, TUnit provides the `@SymbolTablePath` annotation for each test class. The overall idea is to provide a set of symbol table models to define all references that are possible and then to create a test model referencing these symbols.

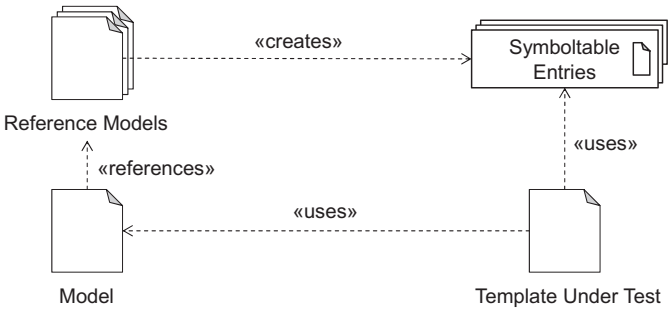


Fig. 8: Overview of the symbol table mocking approach.

In order to extend or mock the symbol table for testing, the testers need to create one or multiple models conforming to the input language of the code generator template, e.g., class diagram language. By annotating a TUnit test class with the `@SymbolTablePath` annotation, the path to the input models that should be used for building the set of symbol table entries is defined. TUnit loads each model and stores all symbols in one symbol table that is provided to the code generator template during execution. An overview of this approach is presented in Figure 8.

This approach of providing symbol table entry information to the template under test is inline with the TUnit's overall approach to separate context information that need to be provided and defining inputs for the template under test. Consequently, context information and in particular symbol table models can be reused for varying inputs.

5.3 Mocking Sub-Template Calls

A TUnit test case may not fail for all defined inputs but the overall code generator may still produce invalid code. This is due to embedded sub-template calls in templates under test. The mentioned example of the invalid code produced by the code generator may happen if the embedded sub-template calls are mocked. In contrast, without pruning the sub-templates, creating the TUnit test may be time-consuming, because all helpers, variables, symbol table entries, and template references need to be considered. Clearly, without pruning any sub-templates, the test coverage, i.e., the amount of templates that are executed in one TUnit test, is higher.

When to prune sub-templates depends on the template and the testing strategy. A testing strategy that can be used is to always try to neglect pruning sub-templates if the sub-

templates do not generate a crucial part of the overall generated code. If the sub-templates are crucial, they should be pruned and tested in a separate TUnit test. Obviously, the term crucial depends on the tester and the context.

To allow a tester to test templates in isolation, i.e., by abstracting away from the results of sub-templates, TUnit provides the annotation `@TemplateSubstitutionPolicy` that has to be defined at the test class level. With this annotation, the tester can configure the strategy on how to mock sub-template calls:

- *Replace with empty*: Every sub-template call is replaced with the empty string. This imitates the situation that no sub-templates are called at all.
- *Replace all with template*: Instead of calling the sub-templates, every time a self-defined template is called. The output of applying this template is inserted instead of the original template.
- *Replace with string*: In this case, a string is defined that is returned instead of the result of calling the sub-templates.
- *Provide method*: This strategy is the most flexible strategy, as it allows for configuring which specific sub-template call is replaced by which specific string or template. This has to be implemented in a method annotated with `@InitSubtemplates`.

If the template substitution policy is not specified for a test class, the sub-template calls are not mocked and the results of the sub-template calls are inserted into the template output as usual.

5.4 Checking Failures with Assertions

One deficiency of the assertion mechanisms presented in Section 4.3 is that the testers have to denote the complete expected output. In case a template generates a large file but only small parts of the output should be checked, applying either of them is too laborious.

In the following, a further variant is proposed, which allows for performing checks for dedicated parts of the AST resulting from the template application. In essence, TUnit provides an API that contains assert methods for different kinds of AST nodes. In the following, a few class diagram specific examples are given:

- `assertHasClass(ASTCompilationUnit, String)`: Ensures that the given compilation unit contains a class with a specific name.
- `assertHasAttribute(ASTClass, String, Type)`: Validates that the given class contains an attribute with the given name and a given type.
- `assertHasMethod(ASTClass, String, Type, List<Type>)`: Ensures that the passed class contains a method with the specified name, return type, and the given list of parameter types.

Java

```
1 //Retrieval of template output omitted here
2 String testOutputPath = ...
3
4 ASTMethodDeclaration actMethodDecl =
5   PartialParsing.parseMethodDeclaration(
6     new File(testOutputPath));
7
8 ASTJavaAssert.assertMethodReturnTypeEquals(
9   actMethodDecl, "void");
10
11 ASTJavaAssert.assertMethodNameEquals(
12   actMethodDecl, "methodName");
13
14 ASTJavaAssert.assertMethodHasParameter(
15   actMethodDecl, "String", "param");
```

List. 5: Example for AST-based API assertions.

List. 5 shows an example for using the AST-based API. As in the previous example, the actual template output needs to be parsed to create the AST. In the course of this, the parser reports an error, if the code does not represent a valid method declaration. Subsequently, it is at first checked, whether the return type of the parsed method declaration equals the expected return type (line 8 to 9). After that, it is checked that the method name of the parsed method declaration equals the expected name (line 11 to 12) and that the method has a particular parameter (line 14 to 15).

The main advantage of this strategy is that it is usually less laborious to apply it compared to the previously introduced assertion mechanisms as the testers do not have to denote the complete expected result string. Furthermore, this strategy is usually less fragile as the test results are not necessarily affected by every single character change. One potential downside is that the offered API focuses on high-level checks. Hence, it is not well suited to check for all kinds of fine-grained details. Moreover, the API is bound to a particular target language. Consequently, a new API has to be provided in case a new target language is used.

In the presented example of the template under test in List. 4, we have not considered the case that a sub-template may generate a file rather than a string that is embedded in the parent template. These generated files can also be checked with TUnit; however, the testers need to manually consider such “side effects” by manually extending the test to consider the generated artifacts.

6 Conclusion and Future Work

The use of code generators in MDD demands for strong testing concepts to develop robust code generators. Most existing approaches to test code generators rely on testing the code generator as a whole, by executing the complete code generator. Testing only selected templates or validating fragments of code is not easily possible in these approaches.

In this paper, we have presented a method and TUnit– an extension of JUnit – as corresponding tool support for testing code generators. It can be employed early in the development of code generators where no complete source code artifacts are generated. Since templates are executed in a context that includes helpers, variables, symbol table references, and template references, TUnit provides means to mock specific parts of this context. TUnit takes input models for a code generator and executes the template under test on selected parts of these models. To validate the template output, three assertion strategies have been presented. First, a string comparison between the actual output and the expected output, which needs to be defined explicitly. Second, an AST comparison based on the input model AST and an expected AST. Third, an AST-based API comparison that allows for executing checks on dedicated parts of the AST that is created from the template output.

Currently, the input model has to be a complete class diagram. In future work we plan to support pieces of class diagrams, e.g. a class only or a method declaration. Besides comparing ASTs to find assertion violations, it is also possible to employ transformation languages. Assuming that a transformation language for the generated language exists [WR11, We12], assertions can be defined by defining patterns that need to be matched in the generated code. If a pattern cannot be found, then the assertion is violated. Otherwise, the assertion is correct. Finally, a general question to be addressed is the efficiency of the proposed approach.

References

[BKS04] Baldan, Paolo; König, Barbara; Stürmer, Ingo: Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems. In: Graph Transformations, volume 3256 of LNCS. Springer Berlin Heidelberg, 2004.

[FR07] France, Robert; Rumpe, Bernhard: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering 2007 at ICSE. IEEE Computer Society, 2007.

[Fr15] FreeMarker Template Language. `http://www.freemarker.org/`, October 2015.

[Gr08] Grönniger, Hans; Krahn, Holger; Rumpe, Bernhard; Schindler, Martin; Völkel, Steven: MontiCore: A Framework for the Development of Textual Domain Specific Languages. In: Companion of the 30th International Conference on Software Engineering. ICSE Companion '08. ACM, 2008.

[Hu11] Hutchinson, John; Whittle, Jon; Rouncefield, Mark; Kristoffersen, Steinar: Empirical Assessment of MDE in Industry. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11. ACM, 2011.

[Jö13] Jörges, Sven: Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach, volume 7747 of LNCS. Springer, 2013.

[JU15] JUnit Website. `http://junit.org/`, October 2015.

[KRV08] Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: MontiCore: Modular Development of Textual Domain Specific Languages. In: Objects, Components, Models and Patterns. volume 11 of Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2008.

[KRV10] Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: MontiCore: A Framework for Compositional Development of Domain Specific Languages. International Journal on Software Tools for Technology Transfer, 12, 2010.

[Li14] Liebel, Grischa; Marko, Nadja; Tichy, Matthias; Leitner, Andrea; Hansson, Jörgen: Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain. In: Model-Driven Engineering Languages and Systems, volume 8767 of LNCS. Springer International Publishing, 2014.

[Ra10] Rajeev, A. C.; Sampath, Prahladavaradan; Shashidhar, K. C.; Ramesh, S.: CoGenTe: A Tool for Code Generator Testing. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10. ACM, 2010.

[Sa08] Sampath, Prahladavaradan; Rajeev, A. C.; Ramesh, S.; Shashidhar, K. C.: Behaviour Directed Testing of Auto-code Generators. In: Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods. SEFM '08. IEEE Computer Society, 2008.

[Sc12] Schindler, Martin: Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, 2012.

[St06] Stürmer, Ingo: Systematic Testing of Code Generation Tools - A Test Suite-oriented Approach for Safeguarding Model-based Code Generation. PhD thesis, Department of Compiler Construction and Programming Languages of the Technical University of Berlin, 2006.

[St07] Stürmer, Ingo; Conrad, Mirko; Doerr, Heiko; Pepper, Peter: Systematic Testing of Model-Based Code Generators. IEEE Transactions on Software Engineering, 33(9), 2007.

[SWC05] Stürmer, Ingo; Weinberg, Daniela; Conrad, Mirko: Overview of Existing Safeguarding Techniques for Automatically Generated Code. In: Proceedings of the 2nd International Workshop on Software Engineering for Automotive Systems. SEAS '05. ACM, 2005.

[We12] Weisemöller, Ingo: Generierung domänenspezifischer Transformationssprachen. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, 2012.

[WR11] Weisemöller, Ingo; Rumpe, Bernhard: A Domain Specific Transformation Language. In: ME 2011 - Models and Evolution. 2011.

[Ze06] Zelenov, Sergey V.; Silakov, Denis V.; Petrenko, Alexander K.; Conrad, Mirko; Fey, Ines: Automatic Test Generation for Model-Based Code Generators. In: Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. ISOLA '06. IEEE Computer Society, 2006.

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Lan-guage to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Business 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Er-fahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): “Heute schon das Morgen sehen“

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT:
Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting. CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik –
Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwick-lung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I^{2}CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)

P-161 Gregor Engels, Dimitris Karagiannis Heinrich C. Mayr (Hrsg.)
Modellierung 2010

P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010

P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration

P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-165 Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, Herwig Unger (Eds.)
10th International Conference on Innovative Internet Community Systems (I^2CS) – Jubilee Edition 2010 –

P-166 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
3. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-167 Robert Krimmer, Rüdiger Grimm (Eds.)
4th International Conference on Electronic Voting 2010
co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-168 Ira Diethelm, Christina Dörge, Claudia Hildebrandt, Carsten Schulte (Hrsg.)
Didaktik der Informatik
Möglichkeiten empirischer Forschungsmethoden und Perspektiven der Fachdidaktik

P-169 Michael Kerres, Nadine Ojstersek Ulrik Schroeder, Ulrich Hoppe (Hrsg.)
DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.

P-170 Felix C. Freiling (Hrsg.)
Sicherheit 2010
Sicherheit, Schutz und Zuverlässigkeit

P-171 Werner Esswein, Klaus Turowski, Martin Juhrisch (Hrsg.)
Modellierung betrieblicher Informationssysteme (MobIS 2010)
Modellgestütztes Management

P-172 Stefan Klink, Agnes Koschmider Marco Mevius, Andreas Oberweis (Hrsg.)
EMISA 2010
Einflussfaktoren auf die Entwicklung flexibler, integrierter Informationssysteme
Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

P-173 Dietmar Schomburg, Andreas Grote (Eds.)
German Conference on Bioinformatics 2010

P-174 Arslan Brömme, Torsten Eymann, Detlef Hühnlein, Heiko Roßnagel, Paul Schmücker (Hrsg.)
perspeGKtive 2010
Workshop „Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen“

P-175 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 1

P-176 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 2

P-177 Witold Abramowicz, Rainer Alt, Klaus-Peter Fähnrich, Bogdan Franczyk, Leszek A. Maciaszek (Eds.)
INFORMATIK 2010
Business Process and Service Science – Proceedings of ISSS and BPSC

P-178 Wolfram Pietsch, Benedikt Krams (Hrsg.)
Vom Projekt zum Produkt
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschafts-informatik (WI-MAW), Aachen, 2010

P-179 Stefan Gruner, Bernhard Rumpe (Eds.)
FM+AM`2010
Second International Workshop on Formal Methods and Agile Methods

P-180 Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, Holger Schwarz (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW)
14. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS)

P-181 Michael Clasen, Otto Schätzel, Brigitte Theuvsen (Hrsg.)
Qualität und Effizienz durch informationsgestützte Landwirtschaft, Fokus: Moderne Weinwirtschaft

P-182 Ronald Maier (Hrsg.)
6th Conference on Professional Knowledge Management
From Knowledge to Action

P-183 Ralf Reussner, Matthias Grund, Andreas Oberweis, Walter Tichy (Hrsg.)
Software Engineering 2011
Fachtagung des GI-Fachbereichs Softwaretechnik

P-184 Ralf Reussner, Alexander Pretschner, Stefan Jähnichen (Hrsg.)
Software Engineering 2011
Workshopband
(inkl. Doktorandensymposium)

P-185 Hagen Höpfner, Günther Specht, Thomas Ritz, Christian Bunse (Hrsg.)
MMS 2011: Mobile und ubiquitäre Informationssysteme Proceedings zur 6. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2011)

P-186 Gerald Eichler, Axel Küpper, Volkmar Schau, Hacène Fouchal, Herwig Unger (Eds.)
11th International Conference on Innovative Internet Community Systems (I^2CS)

P-187 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
4. DFN-Forum Kommunikations-technologien, Beiträge der Fachtagung 20. Juni bis 21. Juni 2011 Bonn

P-188 Holger Rohland, Andrea Kienle, Steffen Friedrich (Hrsg.)
DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. 5.–8. September 2011, Dresden

P-189 Thomas, Marco (Hrsg.)
Informatik in Bildung und Beruf INFOS 2011
14. GI-Fachtagung Informatik und Schule

P-190 Markus Nüttgens, Oliver Thomas, Barbara Weber (Eds.)
Enterprise Modelling and Information Systems Architectures (EMISA 2011)

P-191 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2011
International Conference of the Biometrics Special Interest Group

P-192 Hans-Ulrich Heiß, Peter Pepper, Holger Schlingloff, Jörg Schneider (Hrsg.)
INFORMATIK 2011
Informatik schafft Communities

P-193 Wolfgang Lehner, Gunther Piller (Hrsg.)
IMDM 2011

P-194 M. Clasen, G. Fröhlich, H. Bernhardt, K. Hildebrand, B. Theuvsen (Hrsg.)
Informationstechnologie für eine nachhaltige Landbewirtschaftung
Fokus Forstwirtschaft

P-195 Neeraj Suri, Michael Waidner (Hrsg.)
Sicherheit 2012
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 6. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)

P-196 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2012
Proceedings of the 11th International Conference of the Biometrics Special Interest Group

P-197 Jörn von Lucke, Christian P. Geiger, Siegfried Kaiser, Erich Schweighofer, Maria A. Wimmer (Hrsg.)
Auf dem Weg zu einer offenen, smarten und vernetzten Verwaltungskultur
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2012

P-198 Stefan Jähnichen, Axel Küpper, Sahin Albayrak (Hrsg.)
Software Engineering 2012
Fachtagung des GI-Fachbereichs Softwaretechnik

P-199 Stefan Jähnichen, Bernhard Rumpe, Holger Schlingloff (Hrsg.)
Software Engineering 2012
Workshopband

P-200 Gero Mühl, Jan Richling, Andreas Herkersdorf (Hrsg.)
ARCS 2012 Workshops

P-201 Elmar J. Sinz Andy Schürr (Hrsg.)
Modellierung 2012

P-202 Andrea Back, Markus Bick, Martin Breunig, Key Pousttchi, Frédéric Thiesse (Hrsg.)
MMS 2012:Mobile und Ubiquitäre Informationssysteme

P-203 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.)
5. DFN-Forum Kommunikations-technologien
Beiträge der Fachtagung

P-204 Gerald Eichler, Leendert W. M. Wienhofen, Anders Kofod-Petersen, Herwig Unger (Eds.)
12th International Conference on Innovative Internet Community Systems (I2CS 2012)

P-205 Manuel J. Kripp, Melanie Volkamer, Rüdiger Grimm (Eds.)
5th International Conference on Electronic Voting 2012 (EVOTE2012)
Co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-206 Stefanie Rinderle-Ma, Mathias Weske (Hrsg.)
EMISA 2012
Der Mensch im Zentrum der Modellierung

P-207 Jörg Desel, Jörg M. Haake, Christian Spannagel (Hrsg.)
DeLFI 2012: Die 10. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V.
24.–26. September 2012

P-208 Ursula Goltz, Marcus Magnor, Hans-Jürgen Appelrath, Herbert Matthies, Wolf-Tilo Balke, Lars Wolf (Hrsg.) INFORMATIK 2012

P-209 Hans Brandt-Pook, André Fleer, Thorsten Spitta, Malte Wattenberg (Hrsg.) Nachhaltiges Software Management

P-210 Erhard Plödereder, Peter Dencker, Herbert Klenk, Hubert B. Keller, Silke Spitzer (Hrsg.) Automotive – Safety & Security 2012 Sicherheit und Zuverlässigkeit für automobile Informationstechnik

P-211 M. Clasen, K. C. Kersebaum, A. Meyer-Aurich, B. Theuvsen (Hrsg.) Massendatenmanagement in der Agrar- und Ernährungswirtschaft Erhebung - Verarbeitung - Nutzung Referate der 33. GIL-Jahrestagung 20. – 21. Februar 2013, Potsdam

P-212 Arslan Brömme, Christoph Busch (Eds.) BIOSIG 2013 Proceedings of the 12th International Conference of the Biometrics Special Interest Group 04.–06. September 2013 Darmstadt, Germany

P-213 Stefan Kowalewski, Bernhard Rumpe (Hrsg.) Software Engineering 2013 Fachtagung des GI-Fachbereichs Softwaretechnik

P-214 Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mit schang, Theo Härder, Veit Köppen (Hrsg.) Datenbanksysteme für Business, Technologie und Web (BTW) 2013 13. – 15. März 2013, Magdeburg

P-215 Stefan Wagner, Horst Lichter (Hrsg.) Software Engineering 2013 Workshopband (inkl. Doktorandensymposium) 26. Februar – 1. März 2013, Aachen

P-216 Gunter Saake, Andreas Henrich, Wolfgang Lehner, Thomas Neumann, Veit Köppen (Hrsg.) Datenbanksysteme für Business, Technologie und Web (BTW) 2013 – Workshopband 11. – 12. März 2013, Magdeburg

P-217 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.) 6. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung 03.–04. Juni 2013, Erlangen

P-218 Andreas Breiter, Christoph Rensing (Hrsg.) DeLFI 2013: Die 11 e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. (GI) 8. – 11. September 2013, Bremen

P-219 Norbert Breier, Peer Stechert, Thomas Wilke (Hrsg.) Informatik erweitert Horizonte INFOS 2013 15. GI-Fachtagung Informatik und Schule 26. – 28. September 2013

P-220 Matthias Horbach (Hrsg.) INFORMATIK 2013 Informatik angepasst an Mensch, Organisation und Umwelt 16. – 20. September 2013, Koblenz

P-221 Maria A. Wimmer, Marijn Janssen, Ann Macintosh, Hans Jochen Scholl, Efthimios Tambouris (Eds.) Electronic Government and Electronic Participation Joint Proceedings of Ongoing Research of IFIP EGOV and IFIP ePart 2013 16. – 19. September 2013, Koblenz

P-222 Reinhard Jung, Manfred Reichert (Eds.) Enterprise Modelling and Information Systems Architectures (EMISA 2013) St. Gallen, Switzerland September 5. – 6. 2013

P-223 Detlef Hühnlein, Heiko Roßnagel (Hrsg.) Open Identity Summit 2013 10. – 11. September 2013 Kloster Banz, Germany

P-224 Eckhart Hanser, Martin Mikusz, Masud Fazal-Baqaie (Hrsg.) Vorgehensmodelle 2013 Vorgehensmodelle – Anspruch und Wirklichkeit 20. Tagung der Fachgruppe Vorgehensmodelle im Fachgebiet Wirtschaftsinformatik (WI-VM) der Gesellschaft für Informatik e.V. Lörrach, 2013

P-225 Hans-Georg Fill, Dimitris Karagiannis, Ulrich Reimer (Hrsg.) Modellierung 2014 19. – 21. März 2014, Wien

P-226 M. Clasen, M. Hamer, S. Lehnert, B. Petersen, B. Theuvsen (Hrsg.) IT-Standards in der Agrar- und Ernährungswirtschaft Fokus: Risiko- und Krisenmanagement Referate der 34. GIL-Jahrestagung 24. – 25. Februar 2014, Bonn

P-227 Wilhelm Hasselbring,
Nils Christian Ehmke (Hrsg.)
Software Engineering 2014
Fachtagung des GI-Fachbereichs Softwaretechnik
25. – 28. Februar 2014
Kiel, Deutschland

P-228 Stefan Katzenbeisser, Volkmar Lotz, Edgar Weippl (Hrsg.)
Sicherheit 2014
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
19. – 21. März 2014, Wien

P-230 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2014
Proceedings of the 13th International Conference of the Biometrics Special Interest Group
10. – 12. September 2014 in Darmstadt, Germany

P-231 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.)
7. DFN-Forum Kommunikationstechnologien
16. – 17. Juni 2014
Fulda

P-232 E. Plödereder, L. Grunske, E. Schneider, D. Ull (Hrsg.)
INFORMATIK 2014
Big Data – Komplexität meistern
22. – 26. September 2014
Stuttgart

P-233 Stephan Trahasch, Rolf Plötzner, Gerhard Schneider, Claudia Gayer, Daniel Sassiat, Nicole Wöhrle (Hrsg.)
DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik
der Gesellschaft für Informatik e.V.
15. – 17. September 2014
Freiburg

P-234 Fernand Feltz, Bela Mutschler, Benoît Otjacques (Eds.)
Enterprise Modelling and Information Systems Architectures
(EMISA 2014)
Luxembourg, September 25-26, 2014

P-235 Robert Giegerich,
Ralf Hofestädt,
Tim W. Nattkemper (Eds.)
German Conference on Bioinformatics 2014
September 28 – October 1
Bielefeld, Germany

P-236 Martin Engstler, Eckhart Hanser, Martin Mikusz, Georg Herzwurm (Hrsg.)
Projektmanagement und Vorgehensmodelle 2014
Soziale Aspekte und Standardisierung
Gemeinsame Tagung der Fachgruppen Projektmanagement (WI-PM) und Vorgehensmodelle (WI-VM) im Fachgebiet Wirtschaftsinformatik der Gesellschaft für Informatik e.V., Stuttgart 2014

P-237 Detlef Hühnlein, Heiko Roßnagel (Hrsg.)
Open Identity Summit 2014
4.–6. November 2014
Stuttgart, Germany

P-238 Arno Ruckelshausen, Hans-Peter Schwarz, Brigitte Theuvsen (Hrsg.)
Informatik in der Land-, Forst- und Ernährungswirtschaft
Referate der 35. GIL-Jahrestagung
23. – 24. Februar 2015, Geisenheim

P-239 Uwe Aßmann, Birgit Demuth, Thorsten Spitta, Georg Püschel, Ronny Kaiser (Hrsg.)
Software Engineering & Management 2015
17.-20. März 2015, Dresden

P-240 Herbert Klenk, Hubert B. Keller, Erhard Plödereder, Peter Dencker (Hrsg.)
Automotive – Safety & Security 2015
Sicherheit und Zuverlässigkeit für automobile Informationstechnik
21.–22. April 2015, Stuttgart

P-241 Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, Wolfram Wingerath (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW 2015)
04. – 06. März 2015, Hamburg

P-242 Norbert Ritter, Andreas Henrich, Wolfgang Lehner, Andreas Thor, Steffen Friedrich, Wolfram Wingerath (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW 2015) – Workshopband
02. – 03. März 2015, Hamburg

P-243 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.)
8. DFN-Forum Kommunikationstechnologien
06.–09. Juni 2015, Lübeck

P-244 Alfred Zimmermann,
Alexander Rossmann (Eds.)
Digital Enterprise Computing
(DEC 2015)
Böblingen, Germany June 25-26, 2015

P-245 Arslan Brömme, Christoph Busch ,
Christian Rathgeb, Andreas Uhl (Eds.)
BIOSIG 2015
Proceedings of the 14th International
Conference of the Biometrics Special
Interest Group
09.–11. September 2015
Darmstadt, Germany

P-246 Douglas W. Cunningham, Petra Hofstedt,
Klaus Meer, Ingo Schmitt (Hrsg.)
INFORMATIK 2015
28.9.-2.10. 2015, Cottbus

P-247 Hans Pongratz, Reinhard Keil (Hrsg.)
DeLFI 2015 – Die 13. E-Learning
Fachtagung Informatik der Gesellschaft
für Informatik e.V. (GI)
1.–4. September 2015
München

P-248 Jens Kolb, Henrik Leopold, Jan Mendling
(Eds.)
Enterprise Modelling and Information
Systems Architectures
Proceedings of the 6th Int. Workshop on
Enterprise Modelling and Information
Systems Architectures, Innsbruck, Austria
September 3-4, 2015

P-249 Jens Gallenbacher (Hrsg.)
Informatik
allgemeinbildend begreifen
INFOS 2015 16. GI-Fachtagung
Informatik und Schule
20.–23. September 2015

P-250 Martin Engstler, Masud Fazal-Baqaie,
Eckhart Hanser, Martin Mikusz,
Alexander Volland (Hrsg.)
Projektmanagement und
Vorgehensmodelle 2015
Hybride Projektstrukturen erfolgreich
umsetzen
Gemeinsame Tagung der Fachgruppen
Projektmanagement (WI-PM) und
Vorgehensmodelle (WI-VM) im
Fachgebiet Wirtschaftsinformatik
der Gesellschaft für Informatik e.V.,
Elmshorn 2015

P-251 Detlef Hühnlein, Heiko Roßnagel,
Raik Kuhlisch, Jan Ziesing (Eds.)
Open Identity Summit 2015
10.–11. November 2015
Berlin, Germany

P-252 Jens Knoop, Uwe Zdun (Hrsg.)
Software Engineering 2016
Fachtagung des GI-Fachbereichs
Softwaretechnik
23.–26. Februar 2016, Wien

P-253 A. Ruckelshausen, A. Meyer-Aurich,
T. Rath, G. Recke, B. Theuvsen (Hrsg.)
Informatik in der Land-, Forst- und
Ernährungswirtschaft
Fokus: Intelligente Systeme – Stand der
Technik und neue Möglichkeiten
Referate der 36. GIL-Jahrestagung
22.-23. Februar 2016, Osnabrück

P-254 Andreas Oberweis, Ralf Reussner (Hrsg.)
Modellierung 2016
2.–4. März 2016, Karlsruhe

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de

Gesellschaft für Informatik e.V. (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into
• seminars
• proceedings
• dissertations
• thematics
current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

Information: http://www.gi.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-648-0

"Modellierung 2016" is the fifteenth event in a conference series focusing on a broad range of modeling topics from a variety of perspectives. With its emphasis on lively discussions and cross-fertilization of academia and industry, it provides a valuable platform to further the state of the art in topics such as modeling foundations, methodologies, applications, and tools. This volume contains contributions from the refereed main program.