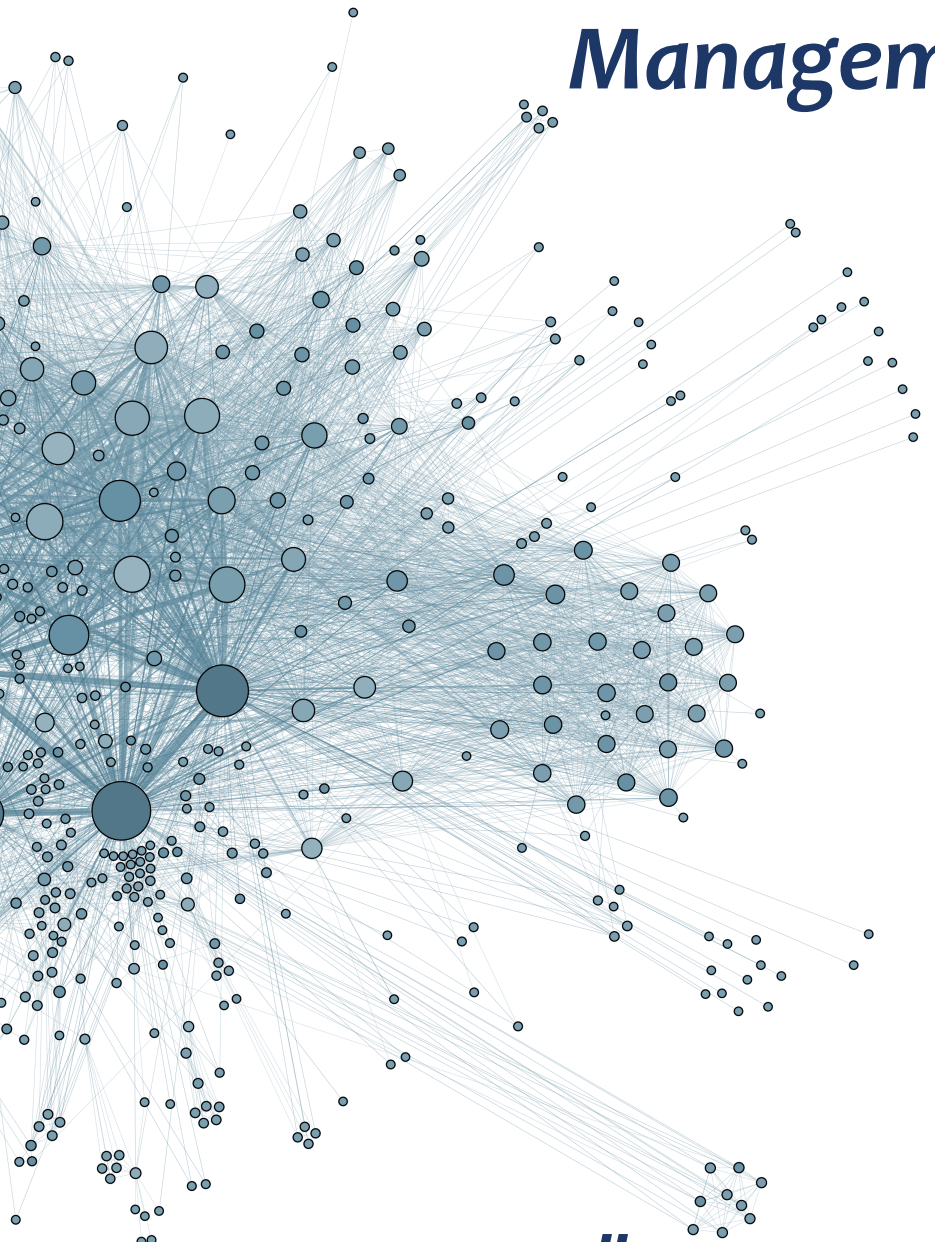
Model analytics and management

Model Analytics and Management



Önder Babur

Model Analytics and Management

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus, prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op woensdag 20 februari 2019 om 16.00 uur

door

Önder Babur

geboren te Ankara, Turkije

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. J.J. Lukkien
promotor:	prof.dr. M.G.J. van den Brand
copromotor:	dr.ir. L.G.W.A. Cleophas
commitee:	prof.dr. M.R.V. Chaudron (Chalmers | University of Gothenburg)
	prof.dr. M. Pechenizkiy
	Prof.Dr.-Ing. I. Schaefer (Technische Universität Braunschweig)
	dr. A. Serebrenik
	prof.dr.ir. B. Tekinerdogan (Wageningen University & Research)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Model Analytics and Management

Önder Babur

Promotor:	prof.dr. M.G.J. van den Brand (Eindhoven University of Technology)
Copromotor:	dr.ir. L.G.W.A. Cleophas (Eindhoven University of Technology)

Additional members of the reading committee:

prof.dr. M.R.V. Chaudron	(Chalmers | University of Gothenburg, Sweden)
prof.dr. M. Pechenizkiy	(Eindhoven University of Technology)
Prof.Dr.-Ing. I. Schaefer	(Technische Universität Braunschweig, Germany)
dr. A. Serebrenik	(Eindhoven University of Technology)
prof.dr.ir. B. Tekinerdogan	(Wageningen University & Research)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
IPA dissertation series 2019-03.

Part of the work in this thesis has been carried out as part of the European Union's FP7 project - Multiscale Modelling Platform: Smart design of nano-enabled products in green technologies - under grant agreement No. 604279.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-4707-4

Cover design: the original graphics of the cover with license CC BY-SA 3.0 is originally downloaded from Internet with the following URL. A minor modification on the color scheme has been performed.

`https://commons.wikimedia.org/wiki/File:SocialNetworkAnalysis.png`

Social network visualization. Published in: Grandjean Martin (2015). "Introduction à la visualisation de données, l'analyse de réseau en histoire" Geschichte und Informatik, 18/19, 2015, 109-128.

Printed by ProefschriftMaken

In loving memory of my mother

Acknowledgements

It has been a long and difficult journey for me to get to the point of completing my PhD. After moving around in Europe — Master's degree and a bit of working in Germany, then research internship in Spain — I was very lucky to get in touch with Mark van den Brand, who offered me a PhD position at Eindhoven University of Technology. I realize that was the time when I finally found the right place to be, the right people to work with and a sense of belonging. I'm grateful to Mark to have such a positive impact on my career and personal life. I'm forever in debt to him for being the mentor he has been all the time, with constant encouragement, guidance and delightful conversations too; especially the ones about food and wine!

The first two years, I had the chance to work with Tom Verhoeff as my daily supervisor in the context of our project MMP. I thank him dearly for bearing with me in the alien territories of multiphysics modelling and simulation. The turning point that led to the successful completion of this thesis, however, is when I decided to change my topic and came up with the idea of model analytics and management. In the beginning, I encountered quite a few obstacles, but I believed in my idea and persevered with my research. Loek Cleophas, who took over the role of my daily supervisor, had a huge impact in my success. I am very happy and grateful having had him by my side. I also thank Alexander Serebrenik, who supported me with a critical stance (despite not being my supervisor) and Ramon Schiffelers for providing me a nice environment at ASML to collaborate.

I have been very happy and felt *at home* at the MDSE group. During my five years, I have had the company of many colleagues, some of whom have already left. I thank them sincerely for constituting the warm welcoming environment in the group, in alphabetical order: Alexander Aroyo, Alexander Fedotov, Aminah Zawedde, Ana-Maria Sutii, Anton Wijs, Arash Khabbaz Saberi, Bogdan Vasilescu, Dan Zhang, Dragan Bosnacki, Erik de Vink, Erik Scheffers, Fei Yang, Felipe Ebert, Frank Peter, Gerard Zwaan, Guilherme Amaral Avelino, Ion Barosan, Jaewon Oh, Jan Friso Groote, Joost Gabriels, Josh Mengerink, Jouke Stoel, Jurgen Vinju, Kees Huizing, Kousar Aslam, Lou Somers, Luc Engelen, Luna Luo, Maarten Manders, Mahmoud Talebi, Margje Mommers-Lenders, Maurice Laveaux, Mauricio Verano Merino, Miguel Botto Tobar, Muhammad Osama, Neda Noroozi, Olav Bunte, Omar Alzuhaiby, Priyanka Karkhanis, Raquel Alvarez Ramirez, Reinier Post, Rick Erkens, Rob Faessen, Rodin Aarssen, Ruurd Kuiper, Sander de Putter, Sangeeth Kochanthara, Sarmen Keshishzadeh, Serguei Roubtsov, Sjoerd Cranen, Thomas Neele, Tim Willemse, Tineke van den Bosch, Ulyana Tikhonova, Weslley Torres, Wieger Wesselink, Yanja Dajsuren and Yuexu Chen.

Throughout my PhD, I have got in touch and collaborated with several colleagues. First of all, I appreciate the contribution of Bedir Tekinerdogan, Mehmet Akşit and Maurice van Keulen for acquiring our community building project, and the committee of 4TU.NIRICT for giving us the opportunity. I am more than happy to have worked with our colleagues from Technische Universität Braunschweig, David Wille, Christoph Seidl and Ina Schaefer. I'm also grateful for Wilbert Alberts for supporting our collaboration at ASML. Last but not least, regarding the workshops MOMA3N and AMMoRe, thanks to the fellow colleagues Michel Chaudron and Davide Di Ruscio for support & co-chairing, and the program committee members for reviewing.

Besides doing research, I have been involved in supervising and working with students. I would like to explicitly thank Aishwarya Suresh, James Hay and Jia Zhang for their direct and indirect contributions in the thesis.

Furthermore, I would like to extend my sincere gratitude to the members of my reading committee for reviewing this dissertation and providing their valuable feedback: Michel Chaudron from Chalmers & University of Gothenburg, Mykola Pechenizkiy, Ina Schaefer from Technische Universität Braunschweig, Alexander Serebrenik, and Bedir Tekinerdogan from Wageningen University & Research.

I also would like to thank SURFSara for their generosity in supplying the computing infrastructure for the experiment performed in this thesis.

Thanks to IPA, I occasionally enjoyed a nice environment with fellow PhD students and researchers in the Netherlands. Moreover, I was lucky to visit a lot of places around the world for conferences and research visits, from Iceland to South Africa and Singapore. I treasure these experiences, which helped me grow on a personal level as well.

Finally, I would like to thank my wife, Burcu, for the love and support she has provided me all this time. Although I am very proud of having written this dissertation, keeping her company is by far the greatest achievement in my life.

Önder Babur
Eindhoven, February 2019

Table of Contents

Acknowledgements **i**

1 Introduction to Model Analytics and Management **1**
1.1 Introduction . . . 1
1.2 The Expanding Universe of MDE . . . 2
1.3 Treating MDE Artifacts as Data . . . 3
1.4 Relevant Domains for Model Analytics and Management . . . 4
1.5 Research Questions . . . 6
1.6 Outline and Origin of Chapters . . . 7

2 SAMOS: A Framework for Model Analytics **11**
2.1 Introduction . . . 11
2.2 Preliminaries: Information Retrieval, Vector Space Model, Clustering . . . 12
2.3 An Architecture for Model Analytics . . . 14
2.4 Case Studies . . . 19
2.5 Discussion . . . 21
2.6 Conclusion . . . 24

3 Structural Comparison of Models **25**
3.1 Introduction . . . 25
3.2 Motivation for Structural Comparison . . . 26
3.3 Extending SAMOS with Structural Features . . . 26
3.4 Case Studies with n-grams . . . 32
3.5 Discussion . . . 34
3.6 Conclusion . . . 38

4 Model Analytics for Variability Mining **39**
4.1 Introduction . . . 39
4.2 Motivating Example and Overall Workflow . . . 41
4.3 Background . . . 45
4.4 Clustering for Variability Mining . . . 47
4.5 Variability Mining for Block-based Languages . . . 49

4.6 Implementation . . . 53
4.7 Case Study . . . 54
4.8 Related Work . . . 59
4.9 Conclusion and Future Work . . . 61

5 Managing a Feature Model Repository 63
5.1 Introduction . . . 63
5.2 Analyzing Feature Models . . . 65
5.3 Case Studies . . . 67
5.4 Discussion and Future Work . . . 73
5.5 Conclusion . . . 74

6 Metamodel Clone Detection with SAMOS 75
6.1 Introduction . . . 75
6.2 Metamodel Clones . . . 77
6.3 Other Model Clone Detector Tools . . . 78
6.4 Using and Extending SAMOS for Clone Detection . . . 81
6.5 Case Studies and Comparative Evaluation . . . 89
6.6 Overall Discussion and Future Work . . . 102
6.7 Related Work . . . 103
6.8 Conclusion . . . 105

7 Model Analytics for Industrial MDE Ecosystems 107
7.1 Introduction . . . 107
7.2 Objectives . . . 108
7.3 MDE Ecosystems at ASML . . . 109
7.4 Model Clones: Concept and Classification . . . 113
7.5 Using and Extending SAMOS for ASOME Models . . . 113
7.6 Case Studies with ASML MDE Ecosystems . . . 116
7.7 Discussion . . . 134
7.8 Related Work . . . 137
7.9 Conclusion and Future Work . . . 140

8 Towards Distributed Model Analytics 141
8.1 Introduction . . . 141
8.2 Background: Apache Spark . . . 142
8.3 Distributed VSM Computation . . . 142
8.4 Preliminary Results and Discussion . . . 144
8.5 Conclusion . . . 146

9 Conclusions 147
9.1 Contributions . . . 147
9.2 Obstacles for Model Analytics and Management . . . 149
9.3 Future Work . . . 150
9.4 The Future of SAMOS as a Mature Open Framework . . . 151

Summary 169

Curriculum Vitae 171

Chapter 1

Introduction to Model Analytics and Management

With the increased adoption of Model-Driven Engineering, the number of related artifacts in use, such as models, metamodels and model transformations, greatly increases. To confirm this, we present quantitative evidence from both academia — in terms of public repositories and datasets — and industry — in terms of large domain-specific language ecosystems. To be able to tackle this dimension of scalability in MDE, we propose to treat the artifacts as data, and apply various techniques — ranging from information retrieval to machine learning — to analyze and manage those artifacts in a holistic, scalable and efficient way.

1.1 Introduction

Model-Driven Engineering (MDE) promotes the use of models, metamodels and model transformations as first-class citizens to tackle the complexity of software systems. As MDE is applied to larger problems, the complexity, size and variety of those artifacts increase. With respect to model size and complexity, for instance, the aspect of scalability has been pointed out by Kolovos et al. [100]. Regarding this aspect, a good amount of research has been done for handling a small number of (possibly very big and complex) models, e.g. in terms of comparison, merging, splitting, persistence or transformation. However, scalability with respect to model variety and multiplicity (i.e. dealing with a large number of possibly heterogeneous models) has so far remained mostly under the radar.

In this thesis, we advocate this aspect of scalability as a potentially big challenge for broader MDE adoption. We highlight evidence and concerns which cross-cut the dichotomies of industry vs. academia and of open source vs. commercial software. We thus show that scalability proves to be an issue overall. Furthermore, we mention several related domains and disciplines as inspiration for tackling scalability, with pointers to some related work. Yet we note the inherent differences of MDE artifacts, e.g. models, compared to common types of data such as natural language text and source code. This might make it difficult to directly apply techniques such as clone detection from other domains to MDE.

1.2 The Expanding Universe of MDE

The aforementioned scalability issue emerges partly due to some recent developments in the MDE community. Firstly, there have been efforts to initiate public repositories to store and manage large numbers of models and related artifacts [32, 169]. Further efforts include mining public repositories for MDE-related items from GitHub, e.g. Eclipse-based MDE technologies [101] and UML models [82] (the Lindholmen dataset). In the latter, the number of UML models can go up to more than 90k. The sheer number of models inevitably calls for techniques for searching, preprocessing (e.g. filtering), analyzing and visualizing the data in a holistic and efficient manner.

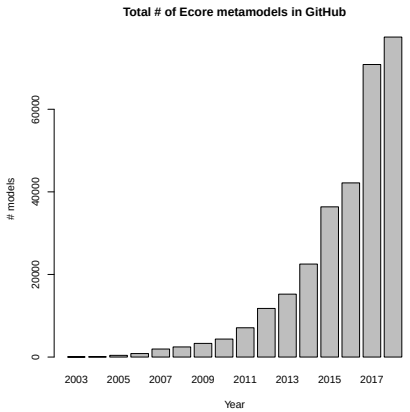
Mini Study: Number of models in public repositories. Kolovos et al. present a study on the use of MDE technologies in GitHub[1] [101]. Among a rich set of empirical results, they report the number of search results for Ecore metamodels in GitHub (as of early 2015) to be $\sim$15k, and show a rapidly increasing trend in the number of commits on MDE-related files. We were triggered by the fact that the same exercise of searching Ecore files, with the query reported in the paper yielded (as of September 2017) more than 67k results; a fourfold increase. We did a preliminary exercise in [26] for Ecore metamodels. Here we repeat the study (as of November 2018) for a wider range of model types; Ecore metamodels, UML models and MPS language models in GitHub, and feature models in S.P.L.O.T.[2]. Figure 1.1 depicts a strong upward trend for all model types across the repositories. Note that these search results might not be 100% accurate (searching based on file extensions and few keywords only), and do not account for *deletions* of models in the repositories (in which case the actual numbers ought to be even higher). We believe these metrics are a very strong indication that there are increasingly more (meta-/)models in public repositories.

MDE in the Industry. Even within a single industry or organization, a similar situation emerges with the increased adoption of MDE. We have been collaborating with high tech companies in the Netherlands. One of those companies maintains a set of MDE-based domain-specific language (DSL) ecosystems. Just a single one of those ecosystems currently contains dozens of metamodels, thousands of models and tens of thousands lines of codes of transformations. With the complete revision history, the total number of artifacts goes up to tens of thousands. Another company, which applies MDE in six different projects, reports a similar collection of thousands of artifacts based on various technologies (e.g. different transformation languages). Similar stories in terms of scale hold for our other industrial partners, with growing heterogeneous sets of artifacts involving multiple domains. Note that for systems with implicit or explicit (e.g. as a Software Product Line) variability, *variants* can be considered another amplifying factor besides *versions* for the total number of MDE artifacts to manage.

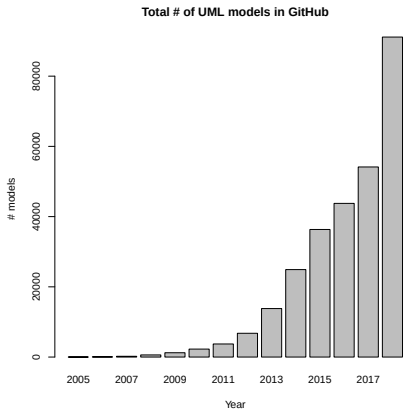
Along with conventional forward engineering approaches, we observe an increasing trend in our partners with legacy software: automated migration into model-driven/-based engineering using process mining and model learning. All the presented facts let us confirm the statement by Brambilla et al. [46] and Whittle et al. [190] that MDE adoption in (at least some parts of) the industry grows quite rapidly, and we conclude that tackling scalability will be increasingly important in the future.

[1] `https://github.com/`

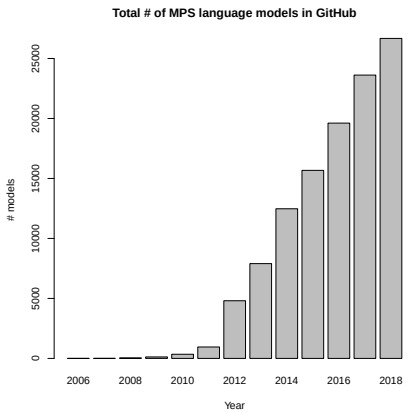
[2] `http://www.splot-research.org/`



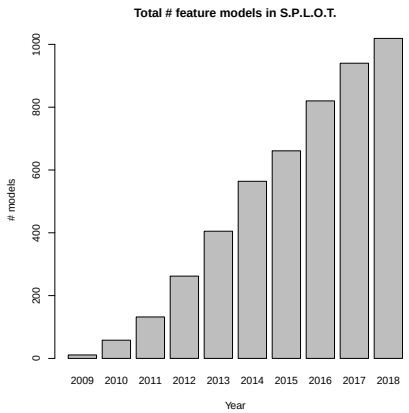
(a) Ecore metamodels in GitHub



(b) UML models in GitHub



(c) MPS language models in GitHub



(d) Feature models in S.P.L.O.T.

Figure 1.1: Total number of various types models in public repositories over the years.

1.3 Treating MDE Artifacts as Data

Based on the observations above, we advocate a perspective where MDE artifacts are treated holistically as data, processed and analyzed with various scalable and efficient techniques, possibly inspired by related domains. Tackling large volumes of artifacts has been commonplace in other domains, such as text mining for natural language text [85], and repository mining for source code [91]. While we might not be able to apply those techniques as-is on MDE-related artifacts, the general data analytics workflow remains as a rough guideline with several steps of data collection, cleaning, integration and transformation, feature engineering and selection, modelling (e.g. as statistical models, neural networks), and finally deployment, exploration and visualization.

To exemplify the different nature of MDE data and hence the different requirements on its analysis, take the problem of clone detection. Clone detection on source code is already several steps away from text mining, as code clone detection usually involves steps such as a language-

specific parsing of the code into abstract syntax trees (AST), normalization of identifiers, and structural transformations [143]. Model clone detection, on the other hand, possess further challenges. To cite Deissenboeck et al., *"Algorithms from code clone detection are only of minor interest for model clone detection, as they usually work on either a linear text or token stream or on the tree structured AST of the code, which both are not transferable to general directed graphs."* [65]. Furthermore, Störrle points out inherent differences of models compared to code, including CASE tool integration and tool-specific representations, internal identifiers and different layouts with no semantic implications, abstract vs. (possibly multiple) concrete syntaxes, etc. [167]. The case of clone detection reinforces our argument that techniques from related domains such as data mining and repository mining might not be directly translatable to the MDE domain.

1.4 Relevant Domains for Model Analytics and Management

Despite the different nature of models as discussed above, we can get inspired by the techniques from other disciplines and try to adapt them for the problems in MDE. As a preliminary overview, in this section we list and discuss several such domains. While there is related MDE research for some of the items on the list, we believe a conscious and integrated mindset would mitigate the challenges for scalable MDE.

Descriptive Statistics. Several MDE researchers have already performed empirical studies on MDE artifacts with a statistical mindset. For instance, Kolovos et al. assesses the use of Eclipse technologies in GitHub, giving related trend analyses [101]. Mengerink et al. present an automated analysis framework on version control systems with descriptive analysis capabilities [120, 121]. Di Rocco et al. perform a correlation analysis on metrics for various MDE artifacts [67]. In all these approaches, authors advocate using metrics to study the nature, interrelation and evolution of MDE artifacts to support, e.g. maintenance tasks. Descriptive statistics could in the most general sense be exploited to gain insight over large numbers of MDE artifacts in terms of general characteristics, patterns, outliers, statistical distributions, dependence, etc.

Information Retrieval. Techniques from information retrieval (IR) can facilitate indexing, searching and retrieving of models, and thus their management and reuse. The adoption of IR techniques on source code dates back to the early 2000s [114], and within the MDE community there has been some recent effort in this direction (e.g. by Bislimovska et al. [43]). Further IR-based techniques involving repository management and model searching scenarios can be found in [22, 35]. In this thesis, we adopt such approaches, which we elaborate and demonstrate in Chapter 2 and the rest of the thesis in detail.

Natural Language Processing. Accurate Natural Language Processing (NLP) is needed to handle realistic models with noisy text content (e.g. in element names), compound words, and synonymy/polysemy. Given the variety of different domains, formalisms/notations and so on, it might be problematic to blindly use NLP tools on models, e.g. just WordNet synonym checking [86] without proper part-of-speech tagging and word sense disambiguation. We need to find the right chain of NLP tools applicable for models (in contrast to source code and documentation), and reporting accuracies and disagreements between tools (along the lines of the recent report in [126] for repository mining). Note that NLP offers further advanced tools, such as

statistical language modelling (see e.g. a recent high-impact study on source code which investigated the *naturalness* of source code [83]) and machine translation, open to be investigated for applications in MDE.

Data Mining. Following the perspective of approaching MDE artifacts as data, we need scalable techniques to extract relevant units of information from models (*features* in data mining jargon), and to discover patterns including clusters, outliers/noise and clones. Several example applications can be found in [22, 35, 17] for domain clustering EMF metamodels, and in [127] for classifying forward vs. reverse engineered UML models. To analyze, explore and eventually make sense of the large datasets in MDE (e.g. the Lindholmen dataset [82]), we can investigate what can be borrowed from comparable approaches in data mining for structured data.

Graph Databases and Graph-based Methods. Given that quite some commonly used models, such as UML, are based on an underlying graph, graph databases can be used to store, query and reason about models. There has already been some effort using graph databases, such as Neo4EMF [38] as a persistence layer for (potentially very big) models, and Mogwaï [61] as a fast and complex querying mechanism for models. Another related idea is presented by Clariso et al. [55], who advocate using graph kernel based methods for several MDE tasks such as model searching and clustering.

Machine Learning. The increasing availability of large amounts of MDE data can be exploited, via machine learning, to automatically infer certain qualities and predictor functions (e.g. performance). There has been a thrust of research in this direction for source code (e.g. for fault prediction [51]), and it would be noteworthy to investigate the emerging needs of the MDE communities and feasibility of such learning techniques for MDE. The approaches in [30] for learning model transformations by examples, and in [31] for automatic model repair using reinforcement learning are some of the few pieces of such work in MDE.

Visualization. We propose visualization and visual analytics techniques to inspect a whole dataset of artifacts (e.g. cluster visualizations in [35], in contrast with visualizing a single big model in [100]) using various features such as metrics and cross-artifact relationships. The goals could range from exploring a repository to analyzing an MDE ecosystem holistically and even studying the (co-)evolution of MDE artifacts.

Distributed/Parallel Computing. With the growing amount of data to be processed, employing distributed and parallel algorithms in MDE is very relevant. There are conceptually related approaches in MDE worthwhile investigating, e.g. distributed model transformations for very large models [39, 49] or model-driven data analytics [80]. Yet we wish to draw attention here to performing computationally heavy data mining or machine learning tasks for large MDE datasets in an efficient way. We also put some effort in this direction by incorporating Apache Spark in our analysis workflow, which will be presented in Chapter 8.

We propose this non-exhaustive list as a preliminary exploitation guideline to help tackling scalability in MDE. Although the aforementioned domains themselves are quite mature on their own, it should be investigated to what extent results and approaches can be transferred into the MDE technical space.

1.5 Research Questions

Having set the scene, we move to our main objectives. Of course, in this thesis we cannot address all of the points above, but rather take a focused (yet relatively generic) perspective. We formulated our main research question as follows.

> **RQ:** *How can we analyze, compare and visualize large sets of models in a generic and scalable way?*

The keywords *generic* and *scalable* are important, and are to be interpreted as, correspondingly, *(conceptually) applicable to multiple types of models* and *able to tackle large sets of models in reasonable execution time*. This central research question is split into three major parts. Each of these questions is addressed in the remainder of this thesis.

One of the notable domains which were listed to get inspiration from, Information Retrieval, advocates the fragmentation of textual data into a wide range of features (e.g. bag of words) for convenient comparison. Fragmentation naturally leads to an approximate representation of the entities, in contrast to e.g. keeping the full entity intact and using holistic pairwise comparison. We follow a similar approach and identify ways of fragmenting models into features to capture model information: model element identifiers, types, attributes, but also encoding parts of the structure of the underlying graph of the models. We further need a general framework for representing the set of models, now fragmented into smaller units, to do analyses on the model level. To investigate the suitable model fragments (i.e. features) and model representations to be used in scalable model analytics, we pose the following research question.

> **RQ_1:** *How (i.e. with which features) can we represent models and their relevant information for scalable analyses?*

Once we have the different types of features extracted from models, the next set of challenges involve the methods for comparing features among themselves, and also for comparing the whole sets of fragments representing each model. Different fragments and types of information might require techniques for accurate comparison. For instance, complex model element names need advanced Natural Language Processing to tackle cases where seemingly different names might be semantically very similar. On the other hand, structural chunks of the model graph would involve considerations on how to compare sets, sequences, ordered or unordered trees and so on. As the next step, the leap from the low level of comparing individual features to the higher level of comparing models raises questions on how to define and search for model similarities while maintaining a balance between scalability and accuracy. These analyses for the whole dataset could then be used holistically to get an overview, identify clusters, outliers and so on. These points are captured in the following research question.

> **RQ_2:** *What techniques can we use to compare models (represented as fragments) in order to discover e.g. similarities, clusters and outliers?*

The techniques and the overall framework we develop allows us to gain insights and see patterns in large sets of models. This valuable information can be exploited in various applications. These range from model recovery, which initiated our research, to model searching, repository management and clone detection, where we performed increasingly more mature case studies. The following research question is related to this aspect of our study.

> **RQ_3:** *How can we exploit the information we acquire through these large scale model analyses, in order to improve the state-of-the-art MDE practices for model analytics and management?*

1.6 Outline and Origin of Chapters

The remainder of this thesis is structured as follows. For each chapter that is based on earlier publications, the origins of the chapter are given. The introduction chapter is partially based on the following paper that appeared in the workshop *Grand Challenges in Modelling* at STAF'17:

> [26] Ö. Babur, L. Cleophas, M. van den Brand, B. Tekinerdogan, M. Aksit. Models, More Models, and Then a Lot More. *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Revised Selected Papers*, page 129-135. Springer, Cham, 2017.

Chapter 2: SAMOS: A Framework for Model Analytics In this chapter, we explain the fundamental concepts underlying our approach, and introduce our framework, SAMOS, with its basic settings ($RQ_{1,2}$). The chapter has the most important role of introducing the big picture in terms of our research, hence answering our central research question. In terms of the applications (RQ_3), we present two case studies involving model searching in GitHub and model management/exploration in a model repository. This chapter is based on the following publications.

> [23] Ö. Babur, L. Cleophas, M. van den Brand. Towards Statistical Comparison and Analysis of Models. *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, page 361-367. Scitepress, 2016.
>
> [22] Ö. Babur, L. Cleophas, M. van den Brand. Hierarchical Clustering of Metamodels for Comparative Analysis and Visualization. *Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA), Held as part of STAF*, page 3-18. Springer, Cham, 2016.
>
> [17] Ö. Babur. Statistical Analysis of Large Sets of Models. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 888-891. ACM, 2016.

Chapter 3: Structural Comparison of Models In this chapter, we extend our basic model analytics framework with a lot of advanced capabilities. The notable extension incorporates model structure for comparison, in terms of linear chunks of n-grams. We provide a quantitative case study to compare the clustering accuracy of n-grams as part of the validation. This chapter mostly addresses $RQ_{1,2}$ and provides a baseline for the future applications. It is based mainly on the following publication:

> [19] Ö. Babur, L. Cleophas. Using n-grams for the Automated Clustering of Structural Models. *Proceedings of the 43rd International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, page 510-524. Springer, 2017.

Chapter 4: Model Analytics for Variability Mining In this chapter, we extend our technique to variability mining from state chart models ($RQ_{1,2}$). We identify outliers and clusters in the input dataset of the variability mining algorithm, and use this information in a preprocessing step to remove and group input data for improving the quality of the output variability models (RQ_3). This chapter is based on the following joint publication with colleagues from TU Braunschweig, who contributed to the variability mining and evaluation parts of the work. Our contribution is in the novel idea, the model clustering and the evaluation parts.

[198] D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, I. Schaefer. Improving Custom-tailored Variability Mining Using Outlier and Cluster Detection. *Science of Computer Programming*, page 62-84. Elsevier, 2018.

Chapter 5: Managing a Feature Model Repository In this chapter, we identify the shortcomings of an uncurated public feature model repository and use the model clustering facilities of SAMOS to find domains and clones in the repository (RQ_3). We extend SAMOS for the feature model type with extended extraction and comparison capabilities needed for feature models ($RQ_{1,2}$). The chapter is based on the following paper in the workshop *Analytics and Mining of Model Repositories (AMMORE)* held at MODELS'18.

[24] Ö. Babur, L. Cleophas, M. van den Brand. Model Analytics For Feature Models: Case Studies for S.P.L.O.T. Repository. *Proceedings of MODELS Workshops co-located with ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, page 787-792. CEUR, 2018.

Chapter 6: Metamodel Clone Detection with SAMOS In this chapter, we extend our technique with additional capabilities ($RQ_{1,2}$) to use in (meta-)model clone detection. Model clone detection has emerged as a major application area for our approach through our studies (RQ_3). We provide mutation and scenario-based evaluation and extensive comparative studies with state-of-the-art model clone detectors. This chapter is based on the following publications.

[18] Ö. Babur. Clone Detection for Ecore Metamodels Using n-grams. *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, page 411-419. Scitepress, 2018.

[27] Ö. Babur, L. Cleophas, M. van den Brand. Metamodel Clone Detection with SAMOS. *Journal of Visual Languages and Computing (JVLC)*, Accepted for publication. Elsevier.

[28] Ö. Babur, L. Cleophas, M. van den Brand. Metamodel Clone Detection with SAMOS (extended abstract). *The 17th edition of the BElgian-NEtherlands software eVOLution symposium (BENEVOL'18)*, Accepted for publication. CEUR, 2018.

Chapter 7: Model Analytics for Industrial MDE Ecosystems In this chapter, we apply our techniques in the context of our industrial partner, ASML. Using and extending our framework SAMOS, we explore a variety of techniques in the domain specific MDE ecosystems at ASML ($RQ_{1,2}$). We present case studies involving clone detection on ASML's data and control models, cross-DSL conceptual analysis and language-level clone detection on multiple ecosystems, and finally architectural analysis on a single ecosystem. We discuss how model analytics can be used to discover insights on MDE ecosystems and opportunities to improve them (RQ_3).

[29] Ö. Babur, A. Suresh, W. Alberts, L. Cleophas, R. Schiffelers, M. van den Brand. Model Analytics for Industrial MDE Ecosystems. *In Model Management and Analytics for Large Scale Systems*, Submitted as a book chapter. Elsevier.

Chapter 8: Towards Distributed Model Analytics In this chapter, we present our preliminary work for plugging SAMOS on top of a distributed computing backend (RQ_2). Thanks to the increased scalability and performance, SAMOS is extendable to cope with bigger datasets and more expensive computations, with an eye towards big data. The following publication captures our approach presented in this chapter.

> [25] Ö. Babur, L. Cleophas, M. van den Brand. Towards Distributed Model Analytics with Apache Spark. *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, page 767-772. Scitepress, 2018.

Chapter 9: Conclusions This final chapter concludes this thesis. We revisit the research questions, discuss obstacles for model analytics and management research, and give directions for the future of our framework and research.

Note that most of the chapters were deliberately kept as similar as possible to how they appear in the corresponding papers, for the sake of understandability, consistency and completeness.

Other Publications Omitted from the Thesis The context of the EU FP7 project involving Multiscale Modelling Platform (MMP) ignited the spark which led to the ideas and research in this thesis. Though along the way, we drifted away from the original MMP domain. Therefore we opted not to put our publications in this thesis as explicit chapters. In our project, we investigated (and attempted to model) MMP frameworks and tools in terms of their architectures, features and so on. Our studies resulted in a publication in the Multiscale Modelling and Simulation workshop held at ICCS'15, a technical report on hundreds of MMP tools. We further contributed a book chapter in Handbook of Software Solutions for ICME, partly thanks to our activities as a member of the working group *Open Simulation Platforms* in *European Materials Modelling Council*.

> [21] Ö. Babur, V. Simulauer, T. Verhoeff, M. van den Brand. A Survey of Open Source Multiphysics Frameworks in Engineering. *Procedia Computer Science - 51*, page 1088-1097. Elsevier, 2015.
>
> [20] Ö. Babur, T. Verhoeff, M. van den Brand. Multiphysics and Multiscale Software Frameworks: An Annotated Bibliography. *Computer science reports; Vol. 1501*, 38 pages. Technische Universiteit Eindhoven, 2015.
>
> [81] A. Hashibon, Ö. Babur, M. Hanzich, G. Houzeaux, B. Patzák. Platforms for ICME. *Handbook of Software Solutions for ICME*, page 533-564. Wiley-Blackwell, 2016.

The research in this thesis resulted in our proposal for a 4TU.NIRICT community building project *Model Management and Analytics* getting funded[3]. We pursued, and are currently continuing on activities which initiated from that project and are published on our portal[4]:

- First Dutch Symposium on Model Management and Analytics, 2017 at Wageningen University and Research, The Netherlands,
- Special Session on Model Management and Analytics, 2018 at the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD),

[3] https://www.4tu.nl/nirict/en/Research/Model-Management-and-Analytics/
[4] https://modelanalytics.wordpress.com/

- First International Workshop on Analytics and Mining of Model Repositories, 2018 at the 21st International Conference on Model Driven Engineering Languages and Systems (MODELS),
- Book on Model Management and Analytics for Large Scale Systems, to be published by Elsevier in 2019.

Chapter 2

SAMOS: A Framework for Model Analytics

Many applications in Model-Driven Engineering involve processing multiple models or metamodels, ranging from model analysis and management to synthesis. Good examples include the comparison and merging of metamodel variants into a common metamodel in domain model recovery, and model versioning. There are numerous techniques that address this challenge in the literature, ranging from graph-based to linguistic ones. Most of these involve pairwise comparison, which might work, e.g. for model versioning with a small number of models to consider. There is, however, the problem of scalability when there is a large number of models to compare. Besides, generally little attention has been given to the initial data analysis, visualization and filtering activities necessary for large scale operations, especially when there are outliers and sub-groupings. We would like to develop a generic approach for model comparison and analysis for large datasets, by using techniques from information retrieval, natural language processing and machine learning. We propose representing models in a vector space model, and applying clustering techniques to analyze, compare and visualize them. We demonstrate our approach on two Ecore datasets: a collection of 50 state machine metamodels extracted from GitHub as top search results; and ∼100 metamodels from 16 different domains, obtained from the AtlanMod Metamodel Zoo.

2.1 Introduction

Model-Driven Engineering (MDE) promotes the use of models and metamodels as first-class artifacts to tackle the complexity of software systems [100]. As MDE is applied to larger problems, the complexity, size and variety of models increase. With respect to model size, the issue of scalability for models has been pointed out by Kolovos et al. [100] as a limiting factor. However, scalability with respect to model variety and multiplicity (i.e. dealing with large and heterogeneous sets of models) is also an important issue, and has been diagnosed by Klint et al. [98] as an interesting aspect to explore. There are many approaches to fundamental operations such as model comparison [159] and matching [99]; applied to problems such as model merging [47], versioning [13] and clone detection [65]; utilizing a wide range of underlying techniques from graph-based to linguistic ones. However those mainly focus on pairwise (or sometimes three-

way, for model versioning) and 'deep' comparison of models in the setting of a very small number of models. Rubin et al. [147] further discuss the inadequacy of pairwise comparison for multiple models, including the effect of comparison order, and propose an N-way model merging algorithm. Indeed, many problems in MDE might involve processing a large number of models. Some examples are domain model recovery from several candidate models [98], metamodel recovery [90], automated domain modeling [139], family mining for Software Product Lines (SPL) from model variants [84, 155], and model clone detection [65].

An initial requirement for large scale model comparison comes from our earlier efforts for domain model recovery. For our project involving the development of a flexible multiphysics and multiscale engineering modelling simulation framework, we were interested in the general case where a common (meta-)model would be reverse engineered out of several candidate (meta-) models. Our scenarios included, for instance, constructing (1) a standardized metadata schema that can support a number of input formats or schemas, and (2) a common metamodel or ontology to orchestrate the interoperability of a heterogeneous set of tools. Our plan was to manually construct (or ideally reverse engineer) those models for individual tools using sources such as design documents and source code; and then perform a manual (or ideally automatic) merge into a big domain model. The study in [20] indicated the overwhelming number of tools in the domain, which made it really difficult to extend our manual model merging efforts in [21] to cover the whole domain. So it became evident that an automated model analysis step would be required at least as a first exploratory step towards automated domain model recovery from multiple sources.

As another concrete setting that occurred to us during the inception phase of our approach, consider the case where a common model (such as a 150% model in SPL terminology) is reverse engineered out of several candidate model variants. We argue that, as the number and variety of input models get larger, the initial data analysis and filtering step gets more relevant. This in turn calls for inspecting the dataset to get an overview and identify potential relations such as proximities, cluster formations and outliers. This information can be used for filtering noisy data, for grouping models, or even for determining the order of processing for pairwise model merging or 150% model generation.

Note that we further found more and more application scenarios for large scale model comparison and analytics, which will be elaborated in the rest of the thesis. To name a few, we identified the need for large-scale model analytics for (1) model searching, (2) model repository management and exploration, (3) clone detection in model repositories and industrial MDE/DSL ecosystems, and (4) empirical studies and repository mining for models.

So in general, we are interested in cases where we need to process a lot of models, in a scalable and efficient way. As an upfront disclaimer, for a big part of this thesis we focus particularly on metamodels; however our approach is generic and thus applicable for the general model comparison and clustering problems. We will demonstrate this in additional case studies in the following chapters; for instance in Chapter 5 for feature models and in Chapter 7 for industrial domain-specific models. The rest of the chapter and thesis uses this convention and refer to both metamodels and models simply as models.

2.2 Preliminaries: Information Retrieval, Vector Space Model, Clustering

We discuss here the underlying concepts of the SAMOS framework [17, 22, 23] inspired by Information Retrieval (IR) and Machine Learning (ML). IR deals with effectively indexing, analyzing, searching and comparing various forms of content including natural language text doc-

uments [113]. As a first step for document retrieval in general, documents are collected and indexed via some unit of representation (*vocabulary* in short): this unit can be bag of words (simply all words, all except stop words, or only some domain-specific terms of interest). Alternatively more complex constructs can be used such as n-grams. N-grams originate from computational linguistics and represent a linear encoding of (text) structure, for example "Julius Caesar" as a single entity rather than each word separately.

Index construction can be implemented using a vector space model (VSM) with the following major components: (1) a vector representation of occurrence or frequency of the vocabulary in a document (named *term frequency* for the latter), (2) optionally *zones* (for instance 'author' or 'title'), (3) weighting schemes such as inverse document frequency (idf), and zone weights, (4) NLP techniques for handling compound terms, detecting synonyms and semantically related words.

As an example, Manning et al. present a very simplistic representation of Shakespeare's plays (originally intended for boolean retrieval) [113]: the vocabulary consists of some important terms, and the vector space is populated with the incidence (i.e. not the frequency) of those terms in the respective plays. Table 2.1 depicts an excerpt from Manning's example.

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
...							

Table 2.1: A simple term incidence matrix representation of Shakespeare's plays (excerpt from [113]).

As exemplified, the VSM allows transforming each document into an n-dimensional vector, thus resulting in an $m \times n$ matrix for m documents. Over the VSM, document similarity can be defined as the distance between vectors, such as Manhattan, Euclidean or cosine, to be chosen considering the underlying problem domain and dataset. Following the example, a very simple dot product of the VSM in Table 2.1 would give us the $m \times m$ pair-wise distance matrix, where (computed for the limited example shown in the table) for instance "Anthony and Cleopatra"·"Julius Caesar"=*3*, "Anthony and Cleopatra"·"Othello"=*1* and "Anthony and Cleopatra"·"The Tempest"=*0*.

Such a distance matrix can in turn be used for identifying similar groups of documents via an unsupervised ML technique called clustering [113, 89]. Among many different clustering methods, there is a major distinction between flat clustering and hierarchical clustering. Flat clustering needs a pre-specified number of clusters and results in a flat assignment of each document into one cluster. k-means is a simple but effective example of this. It aims to identify cluster centers and minimizes the residual sum of (squares of) distances of the points assigned to each cluster. Hierarchical clustering, on the other hand, does not require a pre-specified number of clusters, and outputs a hierarchy of proximities; it thus is more flexible and informative than flat clustering. Hierarchical agglomerative clustering (HAC) builds a nested tree structure (*dendrogram*) of the data points representing proximities and potential clusters, which is suitable for visualization and manual inspection. The HAC algorithm calculates the pairwise distances

of all the points in the dataset. In a bottom-up manner, it starts with each data point in a separate cluster and recursively merges similar points/clusters into bigger clusters. There is a further parameter for HAC for determining how this merge is decided with respect to the inter-cluster distance: single-link assumes cluster distance is the maximum similarity of any individual points in two clusters, while complete and average-link are the correspondingly minimum and average similarity. HAC will be used in SAMOS as we will see in the next section. In summary, the whole process described in this section is what we refer to as a traditional workflow of document clustering in IR.

2.3 An Architecture for Model Analytics

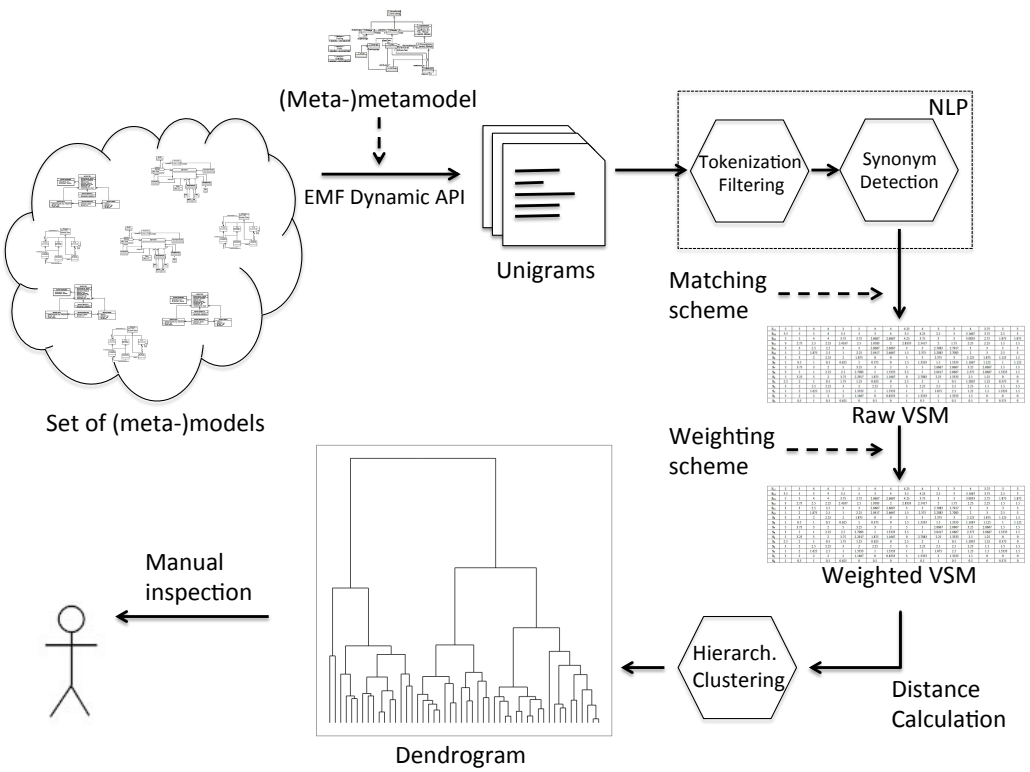


Figure 2.1: Overview of our architecture for SAMOS.

We present our approach and the framework SAMOS (Statistical Analysis of MOdelS), which is a state-of-the-art tool for large-scale analysis of models [17]. We treat models as data, and apply the IR document clustering workflow (introduced in Section 2.2) to models. In this section, we elaborate our approach by a small example. A simple pipeline architecture for our approach is given in Figure 2.1 with the main components as:

1. Data input

   (a) Obtaining a set of models with the same type, e.g. Ecore metamodels in this chapter, to be analyzed,.

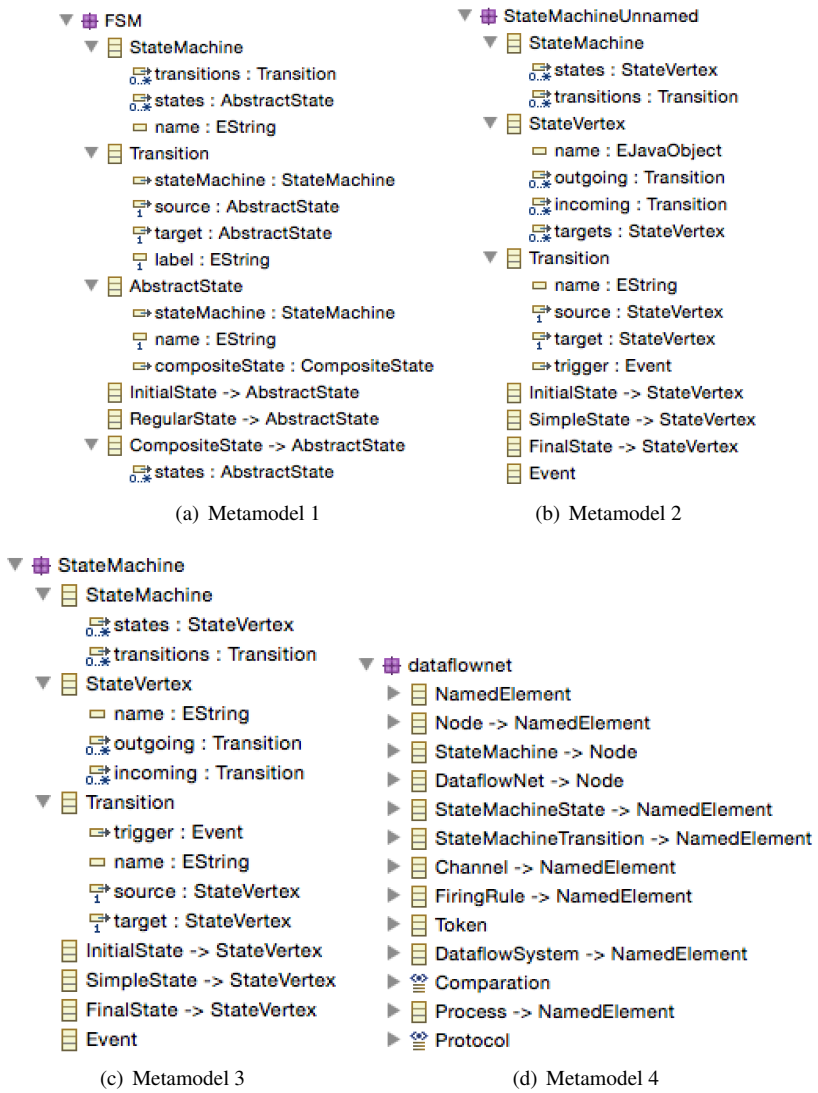
2. Creating VSM representation
   (a) (Feature Extraction) Generating the unigram vocabulary (i.e. the element identifiers) from the input metamodels and the unigram types (similar to zones in IR) from the meta-metamodel (the generic Ecore meta-metamodel in our case, rather than lower level domain-specific ones),
   (b) (Extraction/Postprocessing) Expanding the unigrams with tokenization, and then filtering e.g. stopwords,
   (c) (Feature Comparison) Detecting synonyms and relatedness amongst tokens,
   (d) (Feature Comparison) Utilizing a synonym and type matching mechanism and threshold,
   (e) (VSM Computation) Utilizing an idf and type-based weighting scheme,
   (f) (VSM Computation) Calculating the term frequency matrix.
3. Statistical Analysis
   (a) (Distance Calculation) Picking a distance measure and calculating the vector distances,
   (b) (Clustering) Applying hierarchical clustering over the VSM,
   (c) (Visualization) Visualizing the resulting dendrogram for manual inspection.

The components in the pipeline can be plugged in and out (e.g. different extractors for different model types) and/or switched on and off as required (e.g. synonym checking disabled for certain types of analysis). These will be seen later in the thesis. For consistency, we will provide a configuration of these components here and in each chapter of the thesis.

A Small Example Dataset. Here we introduce a small dataset of Ecore-based metamodels. Ecore is a language used by Eclipse Modelling Framework (EMF) as a common meta-metamodel to support further language development and Eclipse-based software for MDE. Conceptually Ecore metamodels are similar to UML class diagrams. We extract a subset of the metamodel elements (see Section 2.3.1) as representative features. Here we gather 4 metamodels related to state machines, selected from our first case study (Section 2.4.1). The dataset, depicted in Figure 2.2, consists of two plain finite state machine (FSM) metamodels; one hierarchical FSM metamodel; and one data flow metamodel.

2.3.1 Representation as VSM

Generating the unigram vocabulary. From the input metamodels and Ecore meta-metamodel, we construct a typed unigram vocabulary. We adopt a *bag of words* representation for the vocabulary, where each item in the vocabulary is considered individually, discarding the context and order. The type information comes from Ecore ENamedElements, i.e. identifiers: we get the set {*EPackage, EDataType, EClass, EAttribute, EReference, EEnum, EEnumLiteral* and *EDataType*}. Next, we use the EMF API (in Java) to recursively go over all the content for each metamodel element to extract the union of unigrams. The first metamodel in Figure 3.1 would yield *Metamodel 1* = { *(EPackage, FSM), (EClass, StateMachine), (EReference, transitions), (EReference, states), (EAttribute, name),* ... }. Note that several parts of Ecore are deliberately not included in the unigram generation such as EAnnotations and OCL constraints. These are hardly relevant in our case studies and might require further techniques; thus they are left as future work.



(a) Metamodel 1 (b) Metamodel 2

(c) Metamodel 3 (d) Metamodel 4

Figure 2.2: Example dataset.

Vocabulary expansion with tokenization, and then filtering. As identifiers in the metamodels typically are compound names (similar to source code identifiers), we apply tokenization and turn compound names into their tokens to include the vocabulary. We use the Identifier Name Tokenization Tool[1] for implementing this functionality. The types of the original identifiers are retained in the tokens. The expansion of *(EClass,StateMachine)* for instance would yield *(EClass,State)* and *(EClass,Machine)* unigrams. Afterwards we apply a set of filters for the tokens: removal of stop words such as *'of'* and *'from'*, removal of overly short tokens (< 3 characters) and ones consisting of only digits. Note that having done this tokenization step, we use term and token interchangeably for this chapter. It is also noteworthy to mention that tok-

[1] https://github.com/sjbutler/intt

enization reduces the vector space for large datasets significantly: e.g. from 7507 to 5842 for case study 2 (Section 2.4.2). This contributes to the scalability of the approach with respect to the growing size of the dataset.

NLP techniques for synonym and relatedness detection. For the synonym and relatedness detection, we use another array of techniques after normalizing all the tokens into lower case. First of all, we use a Porter Stemmer (Java implementation[2]) for comparing word stems (e.g. *'located', 'location'* and *'locations'* have the common stem *'locat'* and therefore are considered synonyms. Next we measure the normalized Levenshtein distances of the tokens, and consider close words (< 0.1 difference) as synonyms. This allows for approximate string matching, tackling e.g. small typos. Finally, tokens which have a WordNet[3] Wu-Palmer (WuP) similarity score [199] above a certain threshold (0.8 for the examples here) are considered synonyms. We use the WS4J Java library[4] for this calculation.

Unigram matching scheme. We further use a type matching and synonym matching scheme. When comparing two typed unigrams, we add a reducing multiplier of 0.5 for non-exact type matches and use the similarity score as a reducing multiplier for synonym matching. As an example a typed unigram *(EAttribute,name)* would yield 1 when matched against itself, while yielding $0.5 * 0.88 = 0.44$ against *(EReference,label)*, where 0.88 is the WordNet WuP similarity score of *'name'* and *'label'*.

Idf and type weighting scheme. The similarity calculation described above gives a score in the range [0,1] for each metamodel-token pair. On top of this, we apply a weighting scheme on the term incidence matrix, which includes two multipliers: an inverse document frequency (idf) and a type (zone) weight. The idf of a term t is used to assign greater weight to rare terms across metamodels. Idf as the normalized log is defined as:

$$idf(t) = log_{10}\left(1 + \frac{\text{\# total metamodels}}{\text{\# metamodels with the term t}}\right) \tag{2.1}$$

Furthermore, a type weight (specific to the model type to be processed) is given to the unigrams representing their semantic importance. We claim, for instance, that classes are semantically more important than attributes, thus deserve a greater weight. We have used this experimental scheme for this chapter:

$$\begin{aligned} typeWeight(t,w) : \{&\text{EPackage} \rightarrow 1.0, \text{EDataType} \rightarrow 0.2, \text{EClass} \rightarrow 1.0, \\ &\text{EReference} \rightarrow 0.5, \text{EAttribute} \rightarrow 0.3, \text{EEnum} \rightarrow 1.0, \\ &\text{EEnumLiteral} \rightarrow 1.0, \text{EOperation} \rightarrow 0.5, \text{EParameter} \rightarrow 0.1\} \end{aligned}$$

A part of the resulting matrix where all the preprocessing steps above have been done, and the term incidences have been multiplied by idf and weights, is given in Table 2.2.

[2] http://tartarus.org/martin/PorterStemmer/
[3] https://wordnet.princeton.edu/
[4] https://github.com/coriane/ws4j

Metamodel	FSM	State	Machine	source	label	Initial	Channels	...
M1	0.35	0.15	0.15	0.09	0.05	0.15	0	...
M2	0	0.15	0.15	0.09	0.05	0.15	0	...
M3	0	0.15	0.15	0.09	0.04	0.15	0	...
M4	0	0.15	0.15	0	0.04	0.15	0.18	...

Table 2.2: Idf and type weighted term incidence matrix.

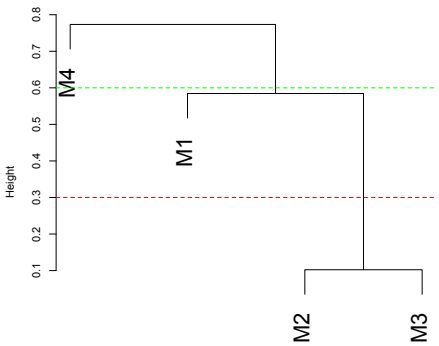


Figure 2.3: Dendrogram of the examples.

	M1	M2	M3
M2	0.61		
M3	0.56	0.10	
M4	0.72	0.81	0.79

Table 2.3: Pairwise distance matrix.

2.3.2 Clustering

Picking a distance measure and calculating the distance matrix. As the next step of our approach, we reduce the metamodel similarity problem into a distance measurement of the corresponding vector representations of metamodels. In earlier preliminary work, we had suggested to pick Manhattan distance [23]. In common natural language text retrieval problems however, cosine distance is used most frequently. Based on the empirical comparisons between the two and the fact that cosine distance is a length normalized metric in the range $[0,1]$ while Manhattan is not, we choose to use cosine distance for the study in this chapter. With p and q being two vectors of n dimensions, cosine distance is defined as:

$$cosineDistance(p,q) = 1 - \frac{p \cdot q}{\| p \| \| q \|} = 1 - \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} \sqrt{\sum_{i=1}^{n} q_i^2}}. \tag{2.2}$$

To be used by the hierarchical clustering, we calculate the pairwise distance matrix of all the models. The distance matrix for the example dataset is given in Table 2.3. We use the `lsa` package in R[5] for this computation [191].

Hierarchical clustering and visualization. We apply agglomerative hierarchical clustering over the VSM to obtain a dendrogram visualization. We used the `hclust` function in the `stats` package [135] with average linkage to compute the dendrogram. The interpretation of this diagram depicted in Figure 2.3 is as follows: the red and green dotted line at heights 0.3 and 0.6 (manually inserted by us) denote horizontal cuts in the dendrogram. Metamodel 4, which stays far above the cut, can be considered as a clear outlier. Depending on the requirements and

[5] `https://www.r-project.org/`

Component		Setting	Description
Extraction	model	Ecore	extractor for Ecore metamodels
	unit	type-name pairs	further units NA at this point
	structure	unigram	structural features NA at this point
	postprocessing	on	token expansion and filtering of stop words
Comparison	basic NLP	on	stemming, Levenshtein
	advanced NLP	on	WordNet semantic relatedness (WuP)
	type matching	relaxed	0.5 multiplier for non-exact types
VSM	frequency	max	max. valued occurrence for similarity
	idf	norm. log	normalized log for idf
	weighting	on	type-based weighting scheme applied
Analysis	distance	cosine	cosine distance (angular)
	clustering	hclust	hierarchical clustering with average linkage
	cut	manual	manual identification of clusters

Table 2.4: SAMOS configuration for the case studies.

interpretation of the user, Metamodels 1-3 can be considered to be in one single cluster (i.e. dendrogram cut at height=0.6) or just Metamodels 2 and 3 (i.e. cut at height=0.3).

2.4 Case Studies

We introduce two case studies to demonstrate the feasibility of our approach. Note that in the strict sense, these are illustrations rather than case studies as formally described by Runeson et al. [148]. Table 2.4 summarizes the SAMOS configuration for the case studies presented in this chapter.

2.4.1 Case Study 1 - GitHub Search Results

Dataset design. For this case study, we searched GitHub[6] on 2016/02/11 for Ecore metamodels using the search terms *'state machine extension:ecore'* and extracted the top 50 results out of 1089 (code) results in total, sorted by *Best Match* criteria. The search facility of GitHub has an internal mechanism for indexing and retrieving relevant text files. Although the intention of this search is to obtain various types of state machine metamodels, we expect to get a heterogeneous dataset, and apply clustering to give an overview of the results.

Objectives. This case study aims to demonstrate the applicability of our approach in a large dataset of a single domain (i.e. state machines), with possible duplicates, outliers, and subdomains. We are interested in large (i.e. > 3 data points) groups of closely similar (e.g. cosine distance < 0.8) metamodels and wish to exclude the outliers. The fact that we obtain metamodels through searching in GitHub also relates to a secondary objective of metamodel searching and exploration (e.g. for reuse, in the sense of traversing a repository/search results and finding the desired metamodels).

Results. Figure 2.4 shows the resulting dendrogram. We have visually identified and labelled the clusters from 1 to 5. Cluster 1 consists of two very similar (distance < 0.1) groups of duplicate

[6] `https://github.com`

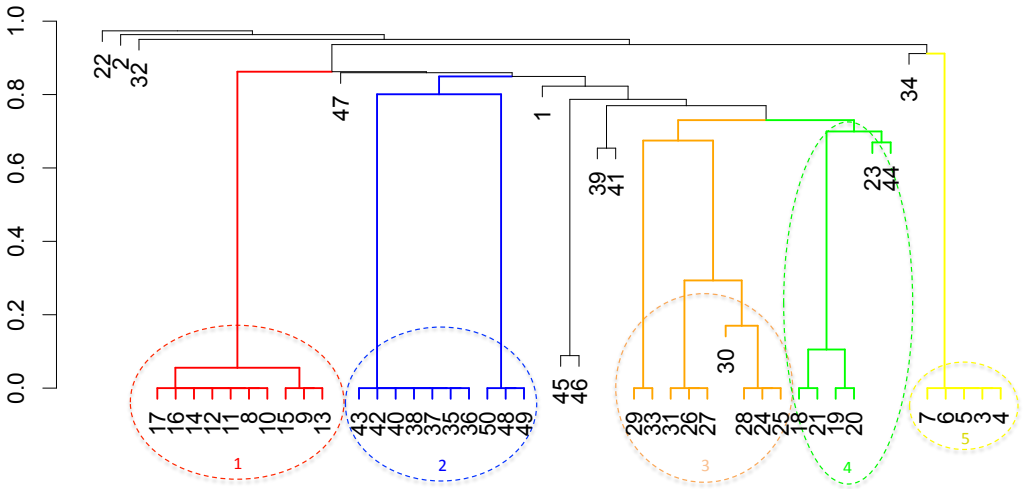


Figure 2.4: Dendrogram of the first dataset.

metamodels (distance = 0) as basic FSMs with states, transitions and associations. In Cluster 2, there are two groups of UML-labelled metamodels with controller elements, triggers, etc. Cluster 3 has metamodels with specializations such as initial and final states, while Cluster 4 has hierarchical state machines with composite states. Metamodel 23 is a false positive; it is labelled NHSM (*non*-hierarchical state machine), nevertheless is erroneously put in Cluster 4. Cluster 5 has duplicate metamodels labelled as AUIML with agents, messages, etc. and is clearly separate from the rest of the clusters. Outliers include a metamodel with identifiers in French (22), a train behaviour metamodel (2), the dataflow metamodel as given in the example dataset (34) and so on. The models 45, 46, 39 and 41 are deliberately not considered as a cluster due to the requirements we set above regarding cluster size and maximum distance.

2.4.2 Case Study 2 - AtlanMod Metamodel Zoo

Dataset design. For this case study, we used a subset of the Ecore metamodels in the AtlanMod Ecore Metamodel Zoo[7]. The Zoo is a collaborative open repository of metamodels in various formalisms including Ecore, intended to be used as experimental material by the MDE community. The repository itself has a wide range of metamodels from different domains; e.g. huge metamodels for programming languages or small class diagram examples for specific problems. We manually selected a subset of 107 metamodels, from 16 different domains. The domain labels are mostly retained as labelled in the repository. Table 2.5 depicts the domain decomposition. The cell below each domain shows the total number of metamodels in that domain, and the corresponding identifiers used in the resulting dendrogram in Figure 2.5.

Objectives. This case study aims to demonstrate the applicability of our approach in a large dataset of multiple domains and subdomains. The domains are chosen to be in a wide range, hence the clustering is meant to show the groups and subgroups in the dataset in a bird's eye point of view. The fact that the metamodels reside in a well-known repository also leads to a side-objective of model repository management and exploration.

[7] `http://web.emn.fr/x-info/atlanmod/index.php?title=Ecore`

Bibliography	Conference	Business Process	Bug Tracker	Multi-agent	ADL
8(1-8)	14(9-22)	6(23-28)	3(29-31)	2(32-33)	15(34-48)
Build Tool	Data Warehouse	Database	Office	Performance	SBVR
5(49-53)	6(54-59)	5(60-64)	10(65-74)	3(75-77)	4(78-81)
Soft. Process	State Machine	Petri Net	Use Case	Total	
3(82-84)	8(85-92)	11(93-103)	4(104-107)	107	

Table 2.5: Number of metamodels in each domain in case study 2

Results. Figure 2.5 shows the resulting dendrogram. We have visually identified and labelled the clusters from 1 to 16. Let us summarize a part of this dendrogram. Cluster 1 (multi-agent) is recognizable as a separate small cluster from the rest of the dataset. Clusters 2 (petri nets) and 3 (state machines) reside as sibling branches. Similarly, clusters 4 (bibliography) and 5 (conference) are clearly detectable as sibling clusters. Cluster 6 and to some extent 8 are a mixture of individual metamodels from different domains, therefore are erroneous according to our initial categorization. Cluster 7 contains build tools. Cluster 9 (database) is in close proximity to the big cluster 10 (Office), the latter of which can be decomposed into two subclusters (left subtree as Word, and right as Excel). Clusters 11-16 correspond to various remaining domains with varying percentages of false positives.

As an external measure of cluster validity, we employ the $F_{0.5}$ measure. Given k as the cluster labels found by our algorithm, l as the reference cluster labels and *cluster pairs* as all the pairs of data points in the same cluster, F_β can be defined as:

$$Precision(k,l) = \frac{|\text{ cluster pairs in } k \cap \text{cluster pairs in } l|}{|\text{ cluster pairs in } k|} \tag{2.3}$$

$$Recall(k,l) = \frac{|\text{ cluster pairs in } k \cap \text{cluster pairs in } l|}{|\text{ cluster pairs in } l|} \tag{2.4}$$

$$F_\beta(k,l) = \frac{(1+\beta^2) * Precision(k,l) * Recall(k,l)}{(\beta^2 * Precision(k,l)) + Recall(k,l)} \tag{2.5}$$

The reason for selecting this measure is that the F_β [112] measure is more common than e.g. purity or the Rand index in the software engineering community, and that we value precision higher than recall; hence the $F_{0.5}$ variant. According to this formula, and using the R package `clusteval` [136] for the co-membership table computation, we obtain an $F_{0.5}$ score of 0.73 for our manual clustering, which can be considered quite high for such a heterogeneous dataset.

2.5 Discussion

This chapter introduces the basics of our approach used in this thesis for model analytics. Based on the two case studies, we confirm our previous claim that a statistical perspective on the comparative analysis and visualization of large datasets seems promising. We make a step towards the handling of large datasets. Using VSM allows a uniform representation of metamodels for statistical analysis, while the accompanying idf and type-based weighting scheme yields a suitable scaling in the vector space. Using a distance measure and hierarchical clustering over VSM, many characteristics and relations among the metamodels, such as clusters, subclusters and outliers, can be analyzed and visualized via a dendrogram. We explore two scenarios, namely model searching and repository exploration, for which we can utilize our approach.

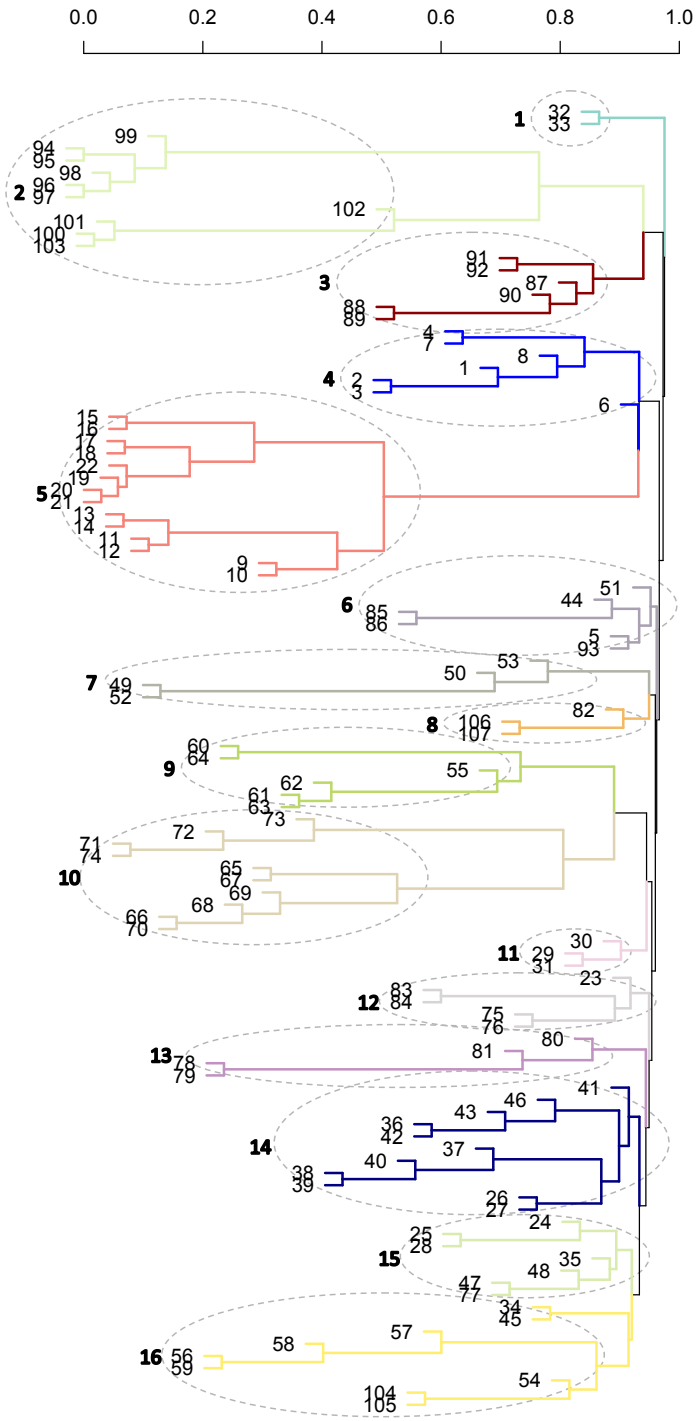


Figure 2.5: Dendrogram of the second dataset.

Particularly for the first case study, it is clearly noticeable that there are distinct outliers and groupings in the search results. This information can be used for instance by a domain model recovery tool to improve the quality of the domain model. Furthermore, the model search functionality, either in GitHub or a specialized model search engine such as [111], can improve the navigation or precision of the search results. The second case study, on the other hand, deals with a heterogeneous set of domains and allows identifying domains, subdomains and also the proximities between related ones. This grouping information can be used for domain model recovery as well as model repository management scenarios.

An advantage of our approach is the scalability and tool support. The algorithm complexities range from linear (e.g. VSM construction) to polynomial (hierarchical clustering) with respect to the size of the dataset and of the metamodels in it. Indeed this technique, and more advanced versions thereof, have already been in widespread use in IR for document retrieval and clustering of large collections of data. Moreover, R provides a plethora of efficient and flexible statistical libraries for analysis. (Meta-)metamodel-based construction of the unigram vocabulary and tokenization provides a good amount of reduction in the vector space, improving over basic IR indexing. Finally we would like to repeat and emphasize that, although we used the term "metamodel" clustering throughout the chapter (because of the datasets we chose), we regard the metamodels as instances of the Ecore meta-metamodel, thus simply as models. Thus we deal with the generic problem of model comparison and clustering.

2.5.1 Threats to Validity

There are several threats to validity for this study. First of all, the NLP techniques employed might not be accurate enough and need to be improved with features such as context-sensitivity and a domain-specific thesaurus. The fact that we regard metamodel identifiers as bag of words and unigrams, thus ignoring structural relations such as containment and inheritance and semantics, could reduce the accuracy and applicability of our approach in some scenarios. Ignoring the multiplicities and modifiers (e.g. abstract) of model elements also might lead to a similar shortcoming. Furthermore, the datasets we used are assembled by us; actual datasets that are used in domain model recovery or SPL extraction should be investigated to compare the results. The visualization and manual inspection approach could limit our approach (as it is now) for larger datasets (e.g. > 1000 items) and further reduction and visualization techniques might be needed. Last but not least, the quantitative comparison of the accuracy of different combinations of parameters and components in virtually every step of our approach, and automation of this process would relieve the user from the effort of trial-and-error exploration of the parameters.

Note that we tackled a number of these threats in our follow-up studies, which are presented in this thesis, notably in Chapters 3 and 6. Examples of these would include; incorporating structural and full attribute information in the models for comparison, automated cluster extraction, extension to various new datasets of large sizes and consisting of new model types, and finally quantitative as well as comparative evaluation of the different settings.

2.5.2 Related Work

Only a few comparison techniques consider multiple input models without pairwise comparisons, such as N-way merging using weighted set packing in [147]. Feature model extraction [158] and concept mining [1] use NLP to cluster concepts. Another technique builds domain ontologies as the intersection of graphs of APIs [137], but does not focus on the scalability dimension of the problem. Metamodel recovery [90] is another approach which assumes a once existing (but somehow lost) metamodel, and does not hold for our scenario. A technique similar to ours is

applied specifically for business process models using process footprints [69], and thus lacks the genericity of our approach. Note that a thorough literature study beyond the technological space of MDE, for instance regarding data schema matching and ontology matching/alignment, is out of scope for this chapter and is therefore omitted.

Clustering, on the other hand, is an older technology; see [113] for a treatment of clustering in the context of information retrieval. It is considered in the software engineering community mostly within a single body of code [104] or model [170]. A recent approach, which we encountered after publishing our early work, is presented by Basciani et al. [33, 34]. They share most of our objectives and some of the techniques we use (i.e. vector representation and clustering), though focusing on repository management. Bislimovska et al. propose information retrieval techniques for indexing and searching WebML models [43].

2.6 Conclusion

In this chapter, we have presented a new perspective on the N-way comparison and analysis of large sets of models for detecting relations such as groupings and outliers among them. We have proposed a generic pipeline architecture, using the IR techniques, and VSM enhanced with NLP techniques to uniformly represent multiple models, and applying hierarchical clustering for comparative analysis and visualization of a large dataset. The pipeline architecture provides a baseline for our approach, allowing new components to be plugged in/out and switched on/off for new application scenarios. We demonstrated our approach on two real datasets; one of top search results from GitHub and another from the AtlanMod Metamodel Zoo. The results, qualitatively for both case studies and quantitatively for the second case study, indicate that our generic and scalable approach is a promising first step for analyzing large datasets of models. The discovered information of groups and outliers effectively serves towards the goal of model management in terms of model searching and repository exploration/management.

Chapter 3

Structural Comparison of Models

Our model analytics approach provides a scalable framework for dealing with large sets of models. In the previous chapter, however, we followed a simplistic approach of representing models with their vertex element information only. In models, particularly in structural models, information is captured not only in model elements (e.g. in names and types) but also in the structural context, i.e. the relation of one element to the others. Our previous approach solved the scalability problem of model analytics while sacrificing structural information. This is in contrast with existing model comparison approaches, which can handle very few (typically two) models by applying sophisticated structural techniques. In this chapter we address both aspects and extend SAMOS with features for incorporating structural context in the form of n-grams. We present a case study, where we compare the n-gram accuracy on two datasets of Ecore metamodels in AtlanMod Zoo: small random samples using up to trigrams and a larger one (∼100 models) up to bigrams. We have observed (1) n-grams do not universally improve accuracy over unigrams, (2) higher n does not lead to monotonically higher accuracy, (3) yet n-grams with $n > 1$ on average perform better than with $n = 1$.

3.1 Introduction

In the previous chapter, we introduced our scalable model analytics framework, SAMOS. While SAMOS focuses on scalability, the previous version (as in Chapter 2) ignores the structural context in models: it extracts model element information (i.e. vertices in the underlying model graph) while discarding the relations (i.e. edges). In this chapter, we aim to extend SAMOS with features for incorporating the structural context in a generic way; acting as a compromise between contextless techniques (i.e. SAMOS in Chapter 2) and expensive pairwise ones such as in [118] and EMFCompare[1]).

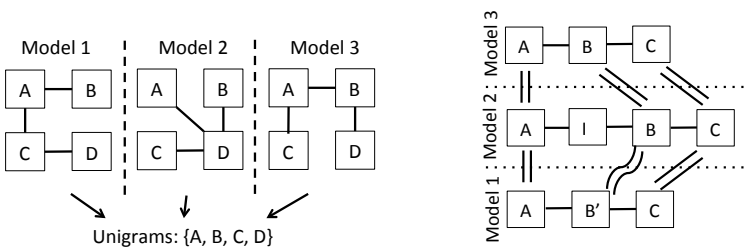
We have extended SAMOS with a type of structural feature, called n-gram, which captures structure in terms of linear chunks. n-grams [112, 179] are used in computational linguistics to build probabilistic models of natural language text, e.g. for estimating the next word given a

[1] `https://www.eclipse.org/emf/compare/` (see [35] for a performance comparison of their clustering approach vs.. EMFCompare

sequence of words, comparing text collections based on their n-gram profiles, or other advanced types of language identification. In essence, n-grams represent a linear encoding of structural context. In the rest of the chapter, we elaborate our extended extraction and comparison schemes for n-grams. For n-grams, we compare the clustering efficiency on two datasets as subsets of the Ecore metamodels in AtlanMod Metamodel Zoo[2]: random samples (20-30 models x 50 runs) using up to trigrams (i.e. $n = 3$) and a larger one (107 models) up to bigrams. We conclude that n-grams lead to higher accuracy on average, though not monotonically with increasing n.

3.2 Motivation for Structural Comparison

Model comparison taking structure information into consideration has been studied in many pairwise comparison approaches (see [159] for an overview). In this section we would like to motivate the problem from the perspective of SAMOS and model clustering. Our previous approach with SAMOS proposes extracting model element names as independent features (i.e. unigrams) to be used in a VSM. This effectively ignores all the structural context of the model. Figure 3.1(a) illustrates one of the shortcomings of using just unigrams for model clustering. An approach as in the previous chapter would treat those three models as the same.



(a) Three models and their unigrams.

(b) Insertion and renaming.

Figure 3.1: Motivating examples for using n-grams.

Another point can be made, given the case that we extract model fragments from three different models (Figure 3.1(b)). The case depicts that the second fragment has the vertex I inserted, while the third fragment has B replaced by its synonym B'. Ideally we would like our clustering technique to treat these three model fragments as strongly similar beyond the unigram similarity of independent $A, B/B'$ and C.

One way of encoding the structural context would be in the form of n-grams of model elements; simpler and cheaper to compare than e.g. subgraphs. Similar ideas has been mentioned in [35] (in the form of character n-grams within the element names), and in [23] (n-grams of model elements from the underlying graph, as we would like to apply in our work), although just for $n = 2$ in the latter. We would like to investigate the general case of using n-grams and their effect on clustering accuracy.

3.3 Extending SAMOS with Structural Features

In this section we describe our extension to SAMOS for clustering structural models based on n-gram representations. We extend and detail the basic feature extraction and comparison scheme

[2] http://web.emn.fr/x-info/atlanmod/index.php?title=Ecore

in Chapter 2. We introduce a generic n-gram setting which allows more sophisticated and accurate comparison with SAMOS. Here we also integrate automated clustering, in contrast to the previously used manual approach.

3.3.1 Models as Labelled Graphs

We consider Ecore metamodels as $M = \langle V, E \rangle$, where V is the set of type-name pair vertices $v = (t_1, n_1)$ and E is the set of edges (t_2, v_s, v_t) with consecutively the edge type (i.e. label of the edge on the underlying labelled graph), source and target of the edges. We consider only a subset of Ecore, and *ENamedEntity* subclasses such as *EPackage, EClass, EAttribute* as vertices; and structural containment, reference type and supertype relations as edges. Thus we omit several parts including *EAnnotations*, OCL constraints and various attributes such as *abstract* for classes, multiplicities for *ETypedElements*, types of *EAttributes* and so on. This restriction yields the domains for the corresponding types: $t_1 \in \{EPackage, EClass, \ldots\}$ and $t_2 \in \{contains, typeof, supertype\}$. Traversing the model and filtering/extracting the desired elements is relatively straightforward using the EMF Java API. Figure 3.2 shows a simple graph representation of an Ecore metamodel.

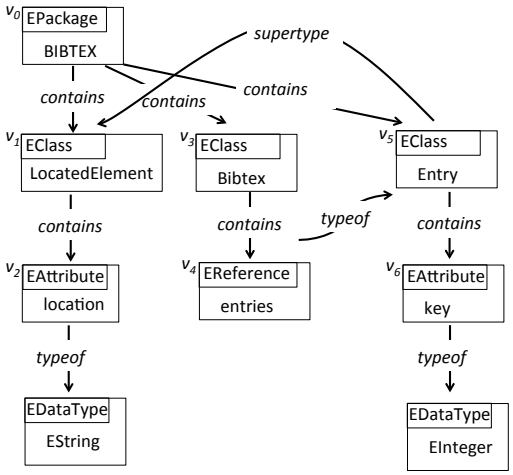


Figure 3.2: Graph representation of an Ecore metamodel.

Regarding whether and how we would like capture the structural context, SAMOS in short offers the following three major settings:

- Ignore the model structure completely, use nodes as is: i.e. the *unigram* setting [22] as in Chapter 2,
- Encode structure in linear chunks: i.e. the *n-gram* (with $n > 1$) setting [19] as discussed in this chapter,
- Encode structure in fixed depth subtrees (omitted here, to be introduced in Chapter 6) [27].

3.3.2 Revisiting Unigrams

Following the simplified graph representation in Figure 3.2, the features as studied in the previous chapter are simply the set of vertices (type-name pairs, i.e. TypedName features in SAMOS) of

the extracted graph: we coin the term *unigram* for those. All the structural information captured in E (the set of edges) is discarded.

Checking the similarity of unigrams is just vertex similarity. As discussed before, the framework implements type/synonym matching, and weighting schemes. For checking the similarity for compound-word names, vocabulary expansion of the unigrams by tokenization (as in Chapter 2) is applicable and effective to be used with regular synonym checking mechanisms for simple words.

To exemplify, follow the graph in Figure 3.2, where the vertices are simplified to domain type and name information with the rest of the attributes being hidden. Examples of type-name pair features to be extracted in our approach are v_0 to v_6.

- v_0 = (EPackage, BIBTEX),
- v_1 = (EClass, LocatedElement),
- v_2 = (EAttribute, location),
- v_3 = (EClass, Bibtex),
- v_4 = (EReference, entries),
- v_5 = (EClass, Entry),
- v_6 = (EAttribute, key).

We have seen how we can extract basic information from metamodels. Next we move on to capturing structural context in features.

3.3.3 Extracting n-grams

We define n-grams using paths of length $n-1$ on the extracted model graph: an n-gram is a sequence of vertices $v_1, \ldots, v_n$ with $n \geq 1$ where for each (v_i, v_{i+1}) there exists some $e \in E$ with type t such that $e = (t, v_i, v_{i+1})$. We further add the restriction that the involved paths have to be *simple* paths, thus having no cycles. With this basic definition, there exists an upper bound for the longest n-gram that can be extracted from non-cyclic paths in the model.

Note that this is a simplified first-attempt formulation of n-grams where edge labels are considered only for the calculation of paths, and are not part of the n-gram itself. We treat edges simply as *relations*, to denote that there is structural association between the vertices. For clone detection in Chapter 6, we extend this simple formulation to include the edges as well.

The above formulation treats the graph in a naive and general-purpose manner. We implemented two alternatives for n-gram extraction in our framework: the regular one and a domain-specific one. In Table 3.1, we list some illustrative results of the regular n-gram extraction from the example in Figure 3.2. We will refer to and use (a slightly modified version of) this regular extraction in some chapters later, notably clone detection (Chapter 6).

As an alternative, we also propose a domain-specific extraction exploiting the actual semantics of the Ecore meta-metamodel. Involving the three types of edges, the following shows how we tackle Ecore models:

- **Rule 1**: Edges of type *contains* are processed regularly in the path traversal.

n	n-grams
2	(v_0, v_1)
	(v_1, v_2)
	(v_5, v_1)
3	(v_0, v_1, v_2)
	(v_3, v_4, v_5)
	(v_5, v_1, v_2)
4	(v_0, v_3, v_4, v_5)
	(v_0, v_5, v_1, v_2)

Table 3.1: Regular n-gram extraction: some examples.

n	n-grams	Rules
3	(v_0, v_1, v_2)	1
	(v_0, v_3, v_4)	1,2
	(v_0, v_3, v_5)	1,2
	(v_3, v_5, v_6)	1,2
	(v_3, v_5, v_2)	1,2,3
4	(v_0, v_3, v_5, v_1)	1,2,3
	(v_0, v_3, v_5, v_2)	1,2,3

Table 3.2: Domain-specific n-gram extraction: some examples.

- **Rule 2**: EReference types can be considered a placeholder for basic associations (e.g. similar to UML associations). We thus fork the path traversal upon encountering *typeof*. One way we include the n-gram up to the EReference vertex and terminate the traversal in order to retain the information encoded by the relation label, if any. The other way we further advance the traversal, jumping over that vertex.
- **Rule 3**: For *supertypes* we exploit the inheritance semantics and fork the path traversal recursively for every *supertype*. This way we cover for instance the implicit associations of a subclass with the attributes of its superclass.

Some examples of the domain specific examples are in turn given in Table 3.2. Note the application of Rules 2,3 for a domain-specific extraction. We have used this extraction in the case studies presented in this chapter.

3.3.4 Defining Vertex and n-gram Similarity

We used a type and name based similarity score to calculate the similarity between the TypedName features in the previous chapter. As aggregate features (such as n-grams) consist of unigrams, we first repeat the vertex comparison for TypedName features with the following multiplicative *vertex similarity* formula:

$$vSim(v_1, v_2) = nameSim(v_1, v_2) * typeSim(v_1, v_2) \tag{3.1}$$

nameSim is the NLP-based similarity between the names; while *typeSim* is between the domain types of the vertices. The framework, as introduced in the previous chapter, allows relaxing the similarity multipliers: e.g. by inserting a reducing multiplier of 0.5 for non-matching types.

Algorithm 1 Maximum similar subsequence.

function $mss(P_1, P_2)$
  $n_1 \leftarrow \text{size}(P_1)$
  $n_2 \leftarrow \text{size}(P_2)$
  declare array $score[n_1+1][n_2+1]$
  declare array $length[n_1+1][n_2+1]$
  for $i = n_1 - 1$ **to** 0 **do** {decrementing}
    for $j = n_2 - 1$ **to** 0 **do** {decrementing}
      if $vSim(V_i, V_j) > 0$ **then**
        $score[i][j] = score[i+1][j+1] + vSim(v_i, v_j)$
        $length[i][j] = length[i+1][j+1] + 1$
      else
        $score[i][j] = max(score[i+1][j], score[i][j+1])$
        $length[i][j] = max(length[i+1][j], length[i][j+1])$
      end if
    end for
  end for
  $m, n \leftarrow i, j$, where $score[i][j] = max(score)$
  return $(score[m][n], length[m][n])$

With the basic similarity calculation of simple features defined, we move to n-grams. Given two n-grams P_1, P_2 with size n containing n vertices $v_1^{1..n}, v_2^{1..n}$ each, we can think of the following similarity schemes:

- strict matching with all vertices equal: 1 if for every $1 \leq i \leq n$, $v_1^i = v_2^i$, 0 otherwise.
- semi-relaxed matching: sum of vertex similarities, times context multiplier:

$$nSim(P_1, P_2) = ctxMult(P_1, P_2) * \sum_{i=1}^{n} vSim(v_1^i, v_2^i) \tag{3.2}$$

$$ctxMult(P_1, P_2) = \frac{1 + |\text{nonzero } vSim \text{ matches between } P_1, P_2|}{1+n} \tag{3.3}$$

- relaxed matching, and using the maximum similar subsequence:

$$nSim'(P_1, P_2) = ctxMult'(P_1, P_2) * score(mss(P_1, P_2)) \tag{3.4}$$

$$ctxMult'(P_1, P_2) = \frac{1 + length(mss(P_1, P_2))}{1+n} \tag{3.5}$$

The function in Equation 3.4, which we call *maximum similar subsequence (mss)*, is a slight modification of the longest common subsequence algorithm, particularly the standard implementation with dynamic programming [40]. We extended the matching of equal elements to incorporate the relaxed vertex similarity schemes. The function is given in Algorithm 1. We don't provide a formal proof for the correctness of the algorithm, but have extensively evaluated and tested it in our case studies and internship assignments through our research project.

Context multipliers given in Equations 3.3 and 3.5 are introduced so that larger percentages of matches contribute to higher similarity. We have implemented variations of this multiplier in

the framework, i.e. normalization (adding 1 to numerator and denominator, inspired by [116]) and power (1 for linear and 2 for quadratic, inspired by the implicit quadratic multiplier in [147]). According to the formulation the multiplier in Equation 3.5 is a normalized linear one. The third scheme (Equations 3.4, 3.5) is used for the rest of the chapter.

3.3.5 Other Modifications to the Framework

Compared to the previous chapter (and hence [22]), one major modification to the framework is in the used NLP techniques. The previous technique uses tokenization and filtering to expand unigrams. For instance, the unigram with the compound name *(EClass, LocatedElement)* would be expanded to two separate unigrams *(EClass, Located)* and *(EClass, Element)*. Afterwards synonym checking is performed on the names of expanded unigrams using algorithms for single-words. While this works efficiently for unigrams, adopting this directly for n-grams has some problems. Expanding an n-gram of size n, with compound-word names of average t tokens leads to a combinatorial explosion (by t^n) of features in the VSM. An example would be the bigram *(EClass, LocatedElement)-(EAtribute, geographicalLocation)* expanding into *{(EClass, Located)-(EAttribute, geographical), (EClass, Located)-(EAttribute, Location),(EClass, Element)-(EAttribute, geographical),(EClass, Element)-(EAttribute, Location)}*. For unigrams, it has been reported in the previous chapter that tokenization helps reducing the vector space as larger datasets tend to have a higher percentage of common tokens. For n-grams, however, this is not the case given the limited dataset: there are not enough common tokenized n-grams in the Ecore dataset used in this chapter and as a result the vector space explodes. For this reason, we have integrated a compound-word vertex similarity measure syn_{multi}. Given two vertices with compound names l_1 and l_2, the similarity is the total sum of maximum synonym matches for each token pair (syn_{simple}, as informally explained in Chapter 2 e.g. with synonym checking and so on), divided by the largest of the token set sizes:

$$syn_{multi}(l_1,l_2) = \frac{\sum_i \underset{j}{argmax}(syn_{single}(T_1^i, T_2^j))}{max(|T_1|,|T_2|)} \tag{3.6}$$

$$T_i = filter(tokenize(l_i)) for\ i = 1,2 \tag{3.7}$$

This technique, supported by a cached lookup for synonyms or an in-memory dictionary, greatly improves the performance of checking synonyms (for n-grams with $n > 1$) over the *'tokenize & expand'* approach used in Chapter 2.

Another parameter we have built into the framework, is the calculation of term frequencies, rather than binary occurrence/incidences. During our experimental runs we have encountered the fact that allowing synonym checks and relaxed type checks leads to multiple non-zero matches across n-grams of different models. Hence two different calculation strategies are integrated into the framework when populating the VSM. Given a model M consisting of n-grams $\{M_1,\ldots,M_n\}$, and its vector space representation S, the value for the vector space cell S_j is:

- **incidence:** $valueAt(M,S_j) = \underset{i}{argmax}(nSim(M_i,S_j))$,
- **frequency:** $valueAt(M,S_j) = \sum_{i=1}^{n} nSim(M_i,S_j)$.

We have empirically evaluated both calculations and observed higher accuracy with the latter strategy in the scope of the experiments in this chapter. The readers should thus assume the latter is applied throughout this chapter. As a brief summary of the extensions, we introduced the feature type n-gram which captures structural information, implemented a domain-specific way of

extracting those from Ecore metamodels, defined similarity calculations on vertices and n-grams, adapted the NLP component for compound words, and integrated frequency-based VSM computation into SAMOS. A final modification to the framework is the use of automatic extraction of clusters from the dendrogram. This will be detailed in the next section, along with the case studies.

3.4 Case Studies with n-grams

In order to quantitatively compare the accuracy of using n-grams with $n = 1$ versus $n > 1$, we have designed two case studies. Before moving on to the case studies themselves, we would like to list exhaustively the parameters of the framework for these experiments. Note that we have deliberately aimed to disable the features which are of relatively less importance for this work; to minimize the overall set of parameters and focus on the ones related to the application of n-grams. The framework settings are detailed below. Readers are also referred to Table 3.3 for a concise summary of these settings.

- n-gram extraction scheme: domain-specific scheme (Section 3.3.3), with $n = \{1,2,3\}$ for the first experiment, and $n = \{1,2\}$ for the second one.
- NLP features: compound-word synonym checking using internal tokenization/filtering (Section 3.3.5) with basic NLP processing such as stemming and Levenshtein distance.
- Type matching: relaxed for model elements with different types, i.e. allowing non-exact type matches.
- n-gram similarity: the above vertex similarity settings (synonym and type matching), with relaxed matching for equal order n-grams and maximum similar subsequence (Section 3.3.4, Equations 3,4).
- VSM calculation: raw VSM with term frequencies (Section 3.3.5).
- Hierarchical clustering: `hclust` function is used with average linkage and cosine distance (from the lsa package[3]) to obtain the dendrogram.

The last step of the framework, i.e. clustering, is enhanced in this work with the automatic extraction or *'cutting'* of the dendrogram. For this we design two scenarios. In scenario 1, the user is assumed to be able to guess the number of clusters in the dataset, say n, with $\pm 20\%$ accuracy. For all the integers in the range: $[floor(0.8*n), ceiling(1.2*n)]$, we apply the standard `cutree` function of R to perform a straight horizontal cut on the dendrogram. As an external measure of cluster validity, we employ the $F_{0.5}$ measure (see previous chapter and [22] for details and the cluster labels for ground truth).

For scenario 2, we assume that the number of clusters cannot be guessed; rather a dynamic cut using the `cutreeDynamic` function in the `dynamicTreeCut`[4] package in R has to be performed. Dynamic cut aims to overcome the simplification of a constant cutoff value, by means of advanced flexible and adaptive cutting algorithms. For this function we use the permutation of the following parameters for `cutreeDynamic`:

- maximum cut height $\in \{0.6, 0.7, 0.8, 0.9\}$; where to cut the tree into subtrees, with height corresponding to the cosine distance in the range [0.0, 1.0].
- minimum cluster size = 2; not to end up with isolated single data points as clusters,
- deep split $\in \{0,1,2\}$; the extent to which subtrees should be further cut into smaller subtrees, i.e. clusters.

[3] `https://cran.r-project.org/package=lsa`
[4] `https://cran.r-project.org/package=dynamicTreeCut`

Component		Setting	Description
Extraction	model	Ecore	extractor for Ecore metamodels
	unit	type-name pairs	further units NA at this point
	structure	n-grams	n-grams with $n = 1,2,3$
	postprocessing	off	no postprocessing
Comparison	basic NLP	on	compound-word similarity w/ internal NLP
	advanced NLP	off	WordNet disabled
	type matching	relaxed	0.5 multiplier for non-exact types
	struct. matching	mss	maximum similar subsequence for n-grams
VSM	frequency	sum	sum of frequencies
	idf	off	idf disabled
	weighting	off	weighting disabled
Analysis	distance	cosine	cosine distance (angular)
	clustering	hclust	hierarchical clustering with average linkage
	cut	automatic	horizontal and dynamic cut

Table 3.3: SAMOS configuration for the case studies.

Run	Unigram	Bigram	Trigram
1	0.693 ± 0.049	0.637 ± 0.154	$\mathbf{0.783 \pm 0.036}$
2	$\mathbf{0.913 \pm 0.064}$	0.891 ± 0.034	0.868 ± 0.049
3	0.796 ± 0.057	$\mathbf{0.799 \pm 0.136}$	0.781 ± 0.121
4	0.542 ± 0.064	0.688 ± 0.185	$\mathbf{0.757 \pm 0.045}$
5	0.576 ± 0.152	0.547 ± 0.118	$\mathbf{0.634 \pm 0.012}$
6	0.691 ± 0.052	0.679 ± 0.094	$\mathbf{0.707 \pm 0.037}$
7	$\mathbf{0.958 \pm 0.027}$	0.956 ± 0.026	0.936 ± 0.044
8	$\mathbf{0.872 \pm 0.180}$	$\mathbf{0.872 \pm 0.180}$	$\mathbf{0.872 \pm 0.180}$
9	0.912 ± 0.08	0.892 ± 0.077	$\mathbf{0.936 \pm 0.035}$
10	0.512 ± 0.12	$\mathbf{0.582 \pm 0.137}$	0.460 ± 0.031
...	...	...	...
Avg.	0.665 ± 0.203	0.682 ± 0.197	$\mathbf{0.700 \pm 0.175}$

Table 3.4: $F_{0.5}$ measures of the runs with regular cut.

3.4.1 Case Study 1 - Random Small Datasets

This case study aims to measure the accuracy of n-grams for relatively small datasets using up to trigrams ($n = 3$). we already specified a subset of the AtlanMod Zoo metamodels, as used previously in Chapter 2. As a reminder, the subset consists of 107 metamodels from 16 different domains (ranging from conference management to state machines). From that subset, we extract further random subsets of smaller sizes. The only restriction is that we pick individual cluster items of size ≥ 2 from each domain/cluster, in order to avoid having a dataset with too many isolated outliers. We run this random procedure 50 times, obtaining 50 datasets of size 20-30. Doing this, we aim to avoid coincidental results for specific corner cases. For each dataset we run the framework with the same settings for unigrams, bigrams and trigrams.

We list the $F_{0.5}$ measures of the runs in the format *mean*$\pm$ *standard deviation* for the random runs: in Table 3.4 for the regular cut scenario and Table 3.5 for the dynamic cut scenario. In both tables, the last row gives the averages over 50 runs. One immediate observation is that bigrams

Run	Unigram	Bigram	Trigram
1	0.693 ± 0.061	$\mathbf{0.771 \pm 0.060}$	0.726 ± 0.074
2	0.706 ± 0.055	0.758 ± 0.093	$\mathbf{0.822 \pm 0.107}$
3	0.548 ± 0.250	0.524 ± 0.142	$\mathbf{0.574 \pm 0.125}$
4	0.464 ± 0.143	$\mathbf{0.693 \pm 0.188}$	0.589 ± 0.135
5	$\mathbf{0.520 \pm 0.141}$	0.469 ± 0.084	0.515 ± 0.047
6	0.694 ± 0.090	0.813 ± 0.068	$\mathbf{0.865 \pm 0.069}$
7	0.671 ± 0.148	$\mathbf{0.938 \pm 0.061}$	0.749 ± 0.058
8	0.887 ± 0.079	$\mathbf{0.958 \pm 0.040}$	0.928 ± 0.097
9	0.742 ± 0.089	$\mathbf{0.880 \pm 0.071}$	0.814 ± 0.025
10	0.517 ± 0.117	$\mathbf{0.542 \pm 0.197}$	0.493 ± 0.137
...	...	...	...
Avg.	0.599 ± 0.200	$\mathbf{0.679 \pm 0.205}$	0.672 ± 0.182

Table 3.5: $F_{0.5}$ measures of the runs with dynamic cut.

and trigrams do not universally improve accuracy over unigrams; counterexamples for this are run 10 in Table 3.4 and run 8 in Table 3.5. Secondly, it also cannot be claimed that picking higher n (e.g. trigrams vs bigrams) leads to monotonically higher accuracy. Indeed the goal of having so many random runs is to come up with an approximate judgment on n-gram accuracy for Ecore metamodels. Bigrams and trigrams perform differently (in comparison with each other) for the two scenarios; nevertheless in the average case for both scenarios, n-grams with $n > 1$ perform better than with $n = 1$.

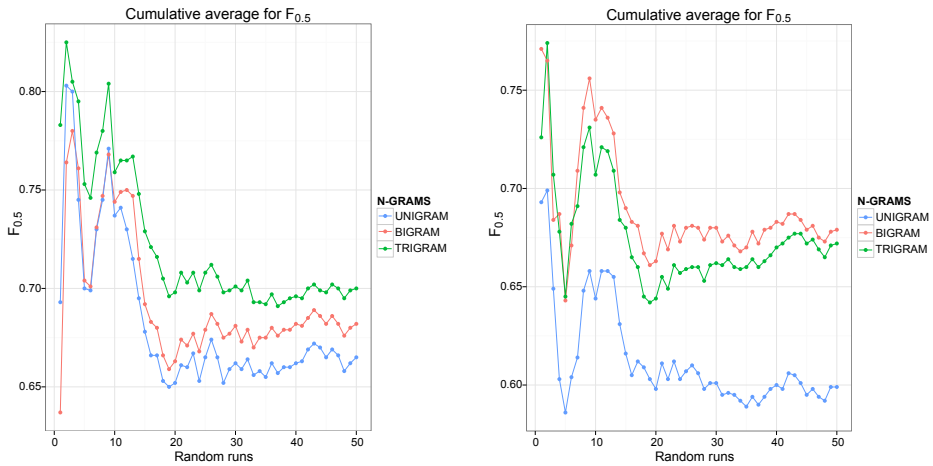
We further supply the line chart of the cumulative mean $F_{0.5}$ measure over the 50 runs in Figure 3.3 for the two scenarios. The points on the diagram correspond to the cumulative mean $F_{0.5}$ values of all the random runs up to k (x axis). This indicates a conclusive *stabilization* after a few runs. This improves our confidence in the measurement, eliminating the chance of e.g. alternating averages over the number of runs.

3.4.2 Case Study 2 - Larger Dataset

With the first case study giving us some insight, we turn to cluster the whole 107-model dataset. We restrict the upper bound for n-grams to bigrams, as trigrams reduce the performance to the point where at least multi-core processing, or high performance computing would be required. Nevertheless, as shown in Figure 3.4, bigrams lead to a considerable increase in the accuracy of the clustering algorithm. The results are given in a boxplot of the $F_{0.5}$ measures with all the parameter permutations for unigrams (left plot) and bigrams (right plot). It is fairly easy to see that bigrams improve the worst case, mean and median; while there is a negligible decrease in the best case (right plot only). Our findings here reinforce our confidence on the average behaviour of bigrams, as pointed out in the first case study.

3.5 Discussion

The two case studies indicate using n-grams with $n > 1$ is a promising technique for incorporating structural context into model clustering. Our technique allows the extraction of model elements together with (part of) their context in the form of n-grams to overcome some difficulties of using just unigrams and losing the context information for model elements (Section 3.2). We



(a) For regular cut. (b) For dynamic cut.

Figure 3.3: Cumulative averages for $F_{0.5}$ over random runs.

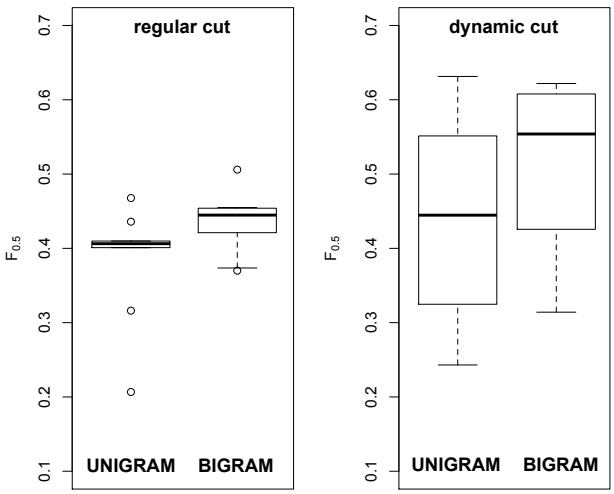


Figure 3.4: Unigram vs bigram $F_{0.5}$ measures.

have relaxed many of the framework parameters and in principle strived for reasoning based on average measurements in order to avoid getting stuck on corner cases and specific parameter settings. The clustering accuracy is shown in two case studies to improve over unigrams on average. Consequently we deduce that:

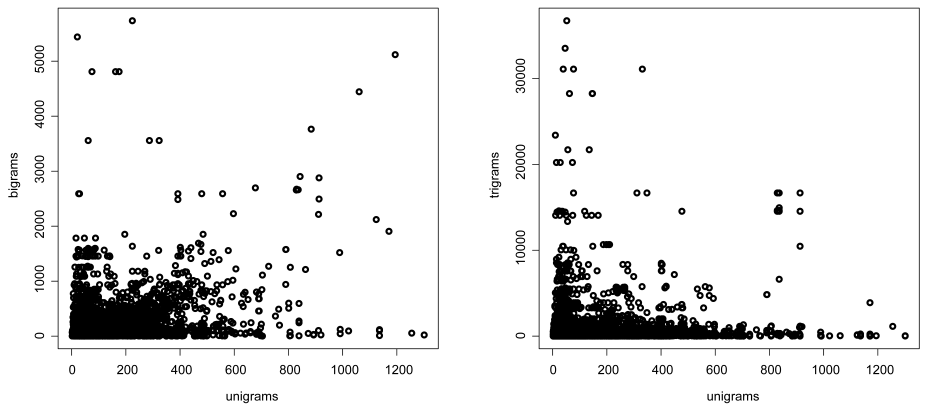
- On average, n-grams with $n > 1$ lead to higher accuracy than $n = 1$.
- The accuracy does not monotonically increase along with increasing n.
- Given the increasing complexity of clustering with larger n, using unigrams remains the most scalable but inaccurate approach, while bigrams can be considered as a safe middle ground for relatively large datasets.
- Depending on the type of the input models, size and nature of the dataset, and for accuracy-oriented tasks, n-grams with $n > 2$ can be employed. However, a preliminary consideration and experimentation should be performed as the accuracy is not guaranteed to increase on average.

Our approach incorporates structural comparison to the N-way model comparison or clustering setting. It can be considered a compromise between the work in [22], which ignores context, and approaches such as [118] which can exploit full structural context for pairwise model comparison. There are further advantages of using this technique, stemming from the underlying framework. For example, the individual steps of the workflow such as the graph-based n-gram extraction and clustering algorithms are generic and extendible for other types of structural models (e.g. UML class diagrams). Furthermore using R brings in strong tool support in terms of clustering, analysis and visualization techniques.

Complexity of Using n-grams. The complexity of hierarchical clustering is $O(m^2 \log m)$[113], with m being the number of data points (i.e. models in our case). The complexity of VSM construction for n-grams is proportional to $|n\text{-}grams|^2 * compare_n$; i.e. the number of extracted n-grams and cost of comparing each n-gram. $compare_n$ is $O(n^2)$ with the maximum similar subsequence implementation in Section 3.3.4. The number of extracted n-grams in turn is proportionate to the size of the input dataset N, average size of models (underlying graph) s and a factor f_n which depends on the n chosen. The formal complexity analysis of f_n would involve measuring the average number of attributes, references and supertypes, plus the graph-theoretic path calculations up to n. This may be difficult to calculate in the domain-specific extraction scheme (Section 3.3.3) we adopt for this chapter. Another approach would be to have a larger and more representative dataset of Ecore models and deduce it empirically using regression on the above mentioned metrics. We leave these as future work, and report here a rough empirical observation on our dataset.

We can safely assume that the number of n-grams for one model is $O(s)$ in the case of unigrams; where s is the model size. For a rough comparison of the n-gram sizes, we crawled 17000+ Ecore models from GitHub[5] and ran our n-gram extraction algorithm on them for $n = 1,2,3$. Figure 3.5 shows the number of bigrams and trigrams (y axis) versus unigrams (x axis) per model. Here we note some simple observations. Unigrams per model tend to be on average in the range of 100s, with bigrams in the 1000s and trigrams growing up to 10000s. There are of course cases with many unigrams and disproportionately few bigrams/trigrams, i.e. presumably flat models with few supertypes/references, and also cases with the opposite; presumably smaller-sized models with complex inheritance hierarchies and cross-references.

[5] https://github.com



(a) # bigrams vs unigrams

(b) # trigrams vs unigrams

Figure 3.5: Empirical observation on the number of n-grams per model.

Threats to Validity. While we aimed for a transparent methodology avoiding coincidental conclusions for corner cases, there are some threats to validity for our work. Firstly, using n-grams should be validated on other and larger datasets of different model types (e.g. UML class diagrams). Secondly, we take only the average accuracies into consideration to conclude the usefulness of using n-grams with $n > 1$, while individual cases (Section 3.4.1) are shown to reduce the accuracy. A meta-analysis is required to find out the reasons, and if possible come up with some heuristics for picking an optimal n considering certain characteristics of the dataset (e.g. size or homogeneity/compactness of clusters and complexity of models in terms of inheritance).

3.5.1 Related Work

Structural comparison has been studied in the context of pairwise model comparison in a lot of studies, e.g. in [118]. These techniques in general develop elaborate pairwise techniques involving graph comparison or isomorphism and aim to reach high accuracy for a small number of models to compare (typically two). On the other hand, there are a few techniques which consider multiple models without pairwise comparisons, such as N-way merging in [147].

Recent approaches such as [23, 22, 35] propose using hierarchical clustering for a large set of (meta-)models. Both use similar Information Retrieval (IR) techniques for extracting term vectors out of models and using various similarity measures such as cosine distance. The use of structural relations among model elements is proposed in [23, 35] encoded as bigrams of model elements; in [35] via the external pairwise comparison operation provided by EMFCompare; while ignored in [22] altogether with the exclusive use of unigrams (i.e. $n = 1$). A final application of n-grams is given by Bislimovska et al. [43] in the context of model indexing and searching.

3.6 Conclusion

In this chapter, we have extended SAMOS to incorporate structural context into clustering. We have indicated a shortcoming of previous approaches, i.e. ignoring the context of model elements, and have proposed a flexible technique, which can be considered as the compromise between structure-agnostic clustering approaches and advanced pairwise structural techniques. We have evaluated our approach on an Ecore dataset from AtlanMod Metamodel Zoo. With carefully devised case studies, avoiding coincidental conclusions for corner cases, we show that n-grams improve the clustering accuracy on average. Picking an $n > 1$ is shown to increase complexity and using larger n is suggested for smaller datasets and precision-oriented tasks, though after preliminary consideration as precision is not guaranteed to increase monotonically along with n. In the end, a more accurate and structure-sensitive setting is desirable in our model analytics studies.

Chapter 4

Model Analytics for Variability Mining

To satisfy demand for customized software solutions, companies commonly use so-called clone-and-own approaches to reuse functionality by copying existing realization artifacts and modifying them to create new product variants. Lacking clear documentation about the variability relations (i.e., the common and varying parts), the resulting variants have to be developed, maintained and evolved in isolation. In previous work, Wille et al. introduced a semi-automatic mining algorithm allowing custom-tailored identification of distinct variability relations for block-based model variants (e.g., MATLAB/Simulink models or statecharts) using user-adjustable metrics. However, variants completely unrelated to other variants (i.e., outliers) can negatively influence the usefulness of the generated variability relations for developers maintaining the variants (e.g., erroneous relations might be identified). In addition, splitting the compared models into smaller sets (i.e., clusters) can be sensible to provide developers separate view points on different variable system features. In the previous chapters, we proposed statistical clustering capable of identifying such outliers and clusters. The contribution of this chapter is the integration of our clustering approach as a preprocessing step to the variability mining algorithm from Wille et al. This allows users to remove outliers prior to executing variability mining on suggested clusters. Using models from two industrial case studies, we show feasibility of the approach and discuss how our clustering can support our variability mining in identifying sensible variability information.[1]

4.1 Introduction

To satisfy demand for customized products, companies often develop *variants* of their software that are specifically tailored to new requirements. These variants form *product families* with largely similar functionality, in which only small parts are newly implemented or slightly adapted compared to other variants [71]. For example, in the automotive domain different variants of

[1] Disclaimer: This is joint work with Wille et al. from Technical University of Braunschweig. The original and fully detailed publication is shortened in this thesis, where we focus on our contribution while only briefly summarizing the parts by our collaborating partners. Please refer to the original publication [198] for the further details.

software for *electronic control units (ECUs)* are needed as additional functionality, such as driver assistance systems or comfort features, can be selected by customers.

Developing each of the software variants in isolation is a tedious task and in most cases infeasible because of the size and complexity of developed systems [71]. Thus, strategies for reuse of existing functionality from previous variants are needed. A common strategy is using so-called *clone-and-own* approaches that copy the code base from an existing variant and modify it to changed requirements. This approach allows easy reuse of implementation solutions for existing features; only the additional functionality has to be realized [71]. However, clone-and-own can have severe consequences during different maintenance tasks as relations between variants are often not documented and no managed reuse strategy exists [71]. Thus, keeping an overview of a growing set of variants becomes almost impossible. For example, duplicate variants might exist that are maintained by different teams completely unaware of the redundant solutions and the unnecessary costs involved. Overall, managing large sets of variants that were created using clone-and-own is a tedious, error-prone and costly task [71, 94, 132]. *Software product lines (SPLs)* allow to introduce managed reuse by maintaining a single code base that allows derivation of all contained variants using suitable generation facilities [56, 60, 133]. One of the benefits is that errors can be fixed in a single location and afterwards affected variants only have to be regenerated. Thus, developers do not have to manually search and fix errors in each variant individually. While these SPLs are a possible solution to clone-and-own related problems, adopting such a reuse strategy without abolishing existing variants is a complex and time-consuming task as *variability relations* (i.e., common and varying parts) between all variants have to be identified to support generation of all variants from a single code base.

Block-based languages, such as *The Mathworks MATLAB/Simulink* [177] or *IBM Rational Rhapsody* [88] statecharts, are commonly used during development of software in domains with high complexity (e.g., the automotive domain). By providing suitable strategies to abstract from complex problems, these languages allow solving these problems on a less complex and more manageable level [64]. While different approaches exist in literature to merge such model variants into a single model containing annotations about the origin of the different parts (e.g., [77, 123, 144, 146]), these approaches do not make variability relations visible to developers. Such relations explicitly categorize parts into *mandatory* parts (i.e., common to all variants), *alternative* parts (i.e., mutually exclusive across all variants), or *optional* parts (i.e., only part of certain variants). To overcome these limitations, Wille et al. introduced their *family mining* approach in previous work, which is capable of semi-automatically reverse-engineering such explicit variability relations for block-based model variants [84, 192, 193, 194]. The approach allows for custom-tailored variability mining as it uses user-adjustable metrics and, thus, provides the possibility to integrate domain knowledge in the executed comparisons between model elements (e.g., knowledge about the influence of certain elements on the model functionality). The identified variability is stored in a single aggregated *150% model* by merging all implementation artifacts from the analyzed models and annotating them with information about their origin (i.e., their parent models) and their explicit variability relations. Overall, the identified variability information allows developers to understand relations between the variants and to get an overview of the existing functionality. While such information eases maintenance of the variants (e.g., variants containing an erroneous part can be easily identified), it most importantly allows to gradually introduce managed reuse strategies from the SPL domain without abolishing variants that were created using clone-and-own approaches [15, 197].

However, as their approach is dependent on the variants that are compared, problems can arise when relations between these possibly large sets of models are unknown. For example, *outliers* (i.e., models completely unrelated to other models) can negatively influence the results, as undesired or unexpected variability might be identified. In addition, it could be more sensible

to split a large set of variants into multiple smaller *clusters* to compare only variants that are functionally close to each other. This way, variability information specifically showing the details of these variants can be provided to developers maintaining them and unnecessary details of unrelated variants can be neglected. In previous work, we developed a statistical clustering technique that is capable of identifying clusters of closely related model variants and ruling out outliers [17, 19, 22, 23]. These were introduced in this thesis in Chapters 2 and 3.

In this chapter, we adapt our clustering techniques to remove outliers prior to executing the variability mining approach of Wille et al. on selected clusters of closely related variants. In particular, we make the following contributions:

- We adapt our clustering technique to identify sensible clusters and to remove outliers from the input models prior to applying the variability mining approach.
- We evaluate whether our adapted clustering technique is able to improve the results of the variability mining algorithm. For our feasibility case study we use industrial-scale *IBM Rational Rhapsody* statechart model variants related to the `body comfort system` (BCS) of a car.

This chapter is structured as follows: Section 4.2 gives a clear motivation of the problem solved by this chapter. Section 4.3 explains background on block-based languages and the existing clustering and variability mining algorithms. Section 4.4 and Section 4.5 describe our cluster and outlier detection approach together with a brief summary of the variability mining approach for block-based languages. Section 4.7 evaluates our solution to demonstrate its feasibility. Section 4.8 discusses related work and Section 4.9 concludes with an outlook to future work.

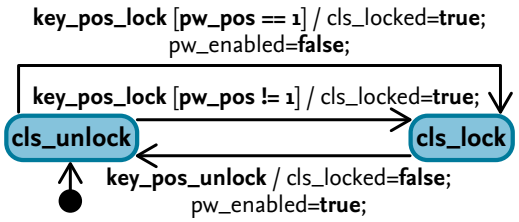
4.2 Motivating Example and Overall Workflow

To give a clear motivation of the problems that we solve in this chapter, we introduce a running example. We first motivate the need for automatic and fine-grained variability mining between large sets of model variants (cf. Subsection 4.2.1) and then discuss challenges with sets of models consisting of multiple subclusters and containing outliers (cf. Subsection 4.2.2). From these observations, we derive our workflow to solve these challenges (cf. Subsection 4.2.3) in the remainder of the chapter.

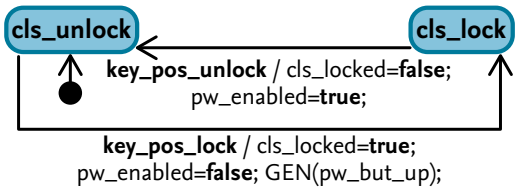
4.2.1 Fine-grained Variability Analysis of Model Variants

To explain the challenges in analyzing the variability between large sets of related model variants, we present in Figure 4.1 two statechart implementations for the *central locking system (`CLS`)* feature of the *body comfort system (`BCS`)* of a car depending on the used *power window (`PW`)*. The `ManPW` variant (cf. Figure 4.1(a)) is equipped with a manual power window that requires the user to push the up or down button until the window is completely opened or closed, whereas the `AutoPW` variant (cf. Figure 4.1(b)) uses a power window automatically closing or opening the window after pushing one of the corresponding buttons.

The only major difference between the implementations manifests itself when the user locks the car and the transition from state `cls_unlock` to state `cls_lock` is executed. The implementation of the `ManPW` variant distinguishes between cases where the window is closed (`pw_pos == 1`) or still open (`pw_pos != 1`). In case the window is already closed, the system also disables the power window (`pw_enabled = false`). Otherwise, the user still can manually close the window after locking the car (e.g., when sitting inside the locked car). In case of the `CLS` implementation for the `AutoPW` variant no distinction is needed and the car automatically disables the power window and closes the windows when locking the car by generating a command (`GEN(pw_but_up)`).



(a) `ManPW` variant of the `CLS` feature



(b) `AutoPW` variant of the `CLS` feature

Figure 4.1: Differing statechart implementations of the *central locking system (`CLS`)* feature for a *body comfort system (`BCS`)* of a car depending on the applied *power window (`PW`)*.

Manually identifying the fine-grained variability relations (i.e., the common and varying elements) between the two implementations of the `CLS` feature (e.g., to apply bugfixes to the variants) might be feasible. However, in industry a large variety of different implementations exists. After the initial clone-and-own process these models evolve independently from each other and developers without initial knowledge about these relations can only reverse-engineer them with large manual effort. In addition, implementations of related features are not directly accessible because they are mostly encapsulated in hierarchical model elements in large implementations consisting of hundreds or thousands of model elements (e.g., the variants presented in Figure 4.1 are part of a *BCS* implementation for a car). In these cases a manual approach fails as time-wise it is infeasible to manually analyze and compare large sets of such complex models in detail. Thus, we argue that automatic variability mining is inevitable to provide developers with detailed variability information of the analyzed variants.

4.2.2 Selecting the Right Variants for Variability Analysis

With the large number of models that were created using clone-and-own approaches, further challenges arise as relations between them are not well documented if at all. As a consequence, selecting the right variants for comparison depends on domain knowledge that might not be available (e.g., after experts left the company or when no documentation exists) and has to be regained using tedious manual techniques. In Figure 4.2 and 4.3, we present two scenarios that show the drawbacks of selecting variants for variability analysis without knowledge of the concrete implementation. Both tables show product variants `V` consisting of different feature combinations that were implemented to realize their functionality (marked with `Xs`). These features are part of the same *BCS* as in the previous example in Figure 4.1 and comprise *exterior mirrors (`EM`)*, *finger protection (`FP`)* for the windows, a *human machine interface (`HMI`)*, a `PW` (either `ManPW` or `AutoPW`), a *remote control key (`RCK`)*, a `CLS`, *heatable exterior mirrors (`Heat`)*, an *alarm system (`AS`)* and corresponding status *`LEDs`*. All variants share the `EM`, `FP`, `HMI` and `PW` features as the core implementation of the *BCS*.

4.2.2.1 Outliers

V	EM	FP	HMI	PW	AutoPW	ManPW	RCK	CLS	Heat	AS	LED	LED_AS	LED_Heat
V1	X	X	X	X		X							
V2	X	X	X	X	X		X	X					
V3	X	X	X	X	X		X	X	X		X		X
V4	X	X	X	X	X		X	X		X	X	X	
V5	X	X	X	X	X		X	X	X	X	X	X	X

Figure 4.2: A basic implementation of a *BCS* forming an *outlier* (variant V1 – highlighted) in contrast to a set of highly related more sophisticated *BCS* implementations (variant V2 – V5).

In Figure 4.2, we show a scenario where two product lines of the *BCS* exist. A very basic variant exists (i.e., V1), only consisting of the core implementation and the ManPW feature. Apart from the core implementation, the other four variants realize a more sophisticated *BCS* (i.e., V2 – V5) for a more expensive car and contain the RCK and CLS together with the possible combinations of the Heat and AS features (plus their corresponding LEDs). In this scenario, V1 can be regarded as an *outlier* because it has almost no relation to the other four variants (apart from the common core implementation). When comparing variants V1 – V5 unexpected or undesired variability might be identified as, for example, developers implementing the RCK and CLS features for the more sophisticated variants V2 – V5 might not be aware of the basic variant V1. As a result, the developers are not only confused by the identified unexpected variability relations but also the maintenance of these variants is hampered. On the one hand, unexpected high-level variability exists because the developers would regard the RCK and CLS features as part of the core implementation. However, in contrast to the developers' expectations, a variability mining algorithm would identify the RCK and CLS features as optional when comparing variants V1 – V5, because V1 does not contain these features. In addition, low-level changes to feature implementations (e.g., their states and transitions) might exist because of necessary feature interactions. For example, the RCK and CLS features might require changes to the HMI feature (e.g., to realize additional control buttons) that are not present in V1. A variability mining algorithm would also identify these low-level changes only present in certain variants and, thus, classifies them as optional. While this is a correct representation of the variability between all developed variants, such variability details confuse developers unaware of these relations. In addition, the identified variability information is bloated with details that are only relevant when considering the variability of all developed variants (i.e., V1 – V5). However, the work of developers might get hampered by details concerning the relation to variant V1 because they need to analyze variability that is irrelevant for maintaining variant V2 – V5. Thus, identifying such possible outliers and ruling them out prior to the variability mining allows to provide developers with information specifically focused on their current task.

4.2.2.2 Clusters

In Figure 4.3, we show another scenario where the manufacturer decided to add two variants V6 and V7 with the ManPW feature in addition to variant V1 from the outlier scenario. Both have the same features as V1 but in addition contain the CLS and the Heat feature. The four variants V2 – V5 are exactly the same as from the outlier scenario. In this scenario, we can identify two *clusters* C1 and C2 of variants that are closely related amongst each other but only have minor similarities across the two clusters (i.e., the common core implementation and similar features across the variants). However, larger differences exist because of feature implementations that

C	V	EM	FP	HMI	PW	AutoPW	ManPW	RCK	CLS	Heat	AS	LED	LED_AS	LED_Heat
C1	V1	X	X	X	X		X							
	V6	X	X	X	X		X		X					
	V7	X	X	X	X		X		X	X				
C2	V2	X	X	X	X	X		X	X					
	V3	X	X	X	X	X		X	X	X		X		X
	V4	X	X	X	X	X		X	X		X	X	X	
	V5	X	X	X	X	X		X	X	X	X	X	X	X

Figure 4.3: Two *clusters* of basic implementations (cluster `C1`: `V1` – `V3`) and more sophisticated implementations (cluster `C2`: `V4` – `V7`) for a *BCS* – the distinct differences are highlighted.

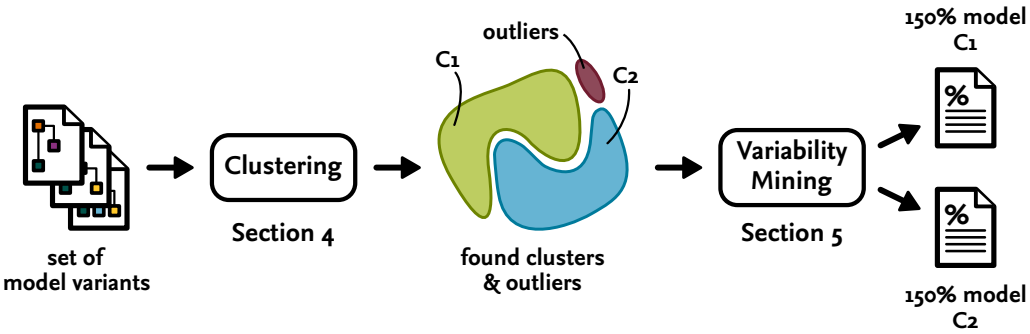


Figure 4.4: Overall workflow for the combined clustering and variability mining approach

are only present in one cluster (highlighted in the table). Similar to the outliers scenario it might be desirable to identify variability for specific single clusters instead of the complete product family. This way only expected relations are shown to the developers of the corresponding product lines. Thus, the developers can concentrate on variability information that is specifically focused on their work with specific variant clusters.

4.2.3 Workflow of the Proposed Solution

From the scenarios discussed in Subsection 4.2.2, we identified the need for:

a) Algorithms to identify fine-grained variability information between related model variants (e.g., created using clone-and-own approaches) to make these relations visible to developers.

b) Clustering and outlier detection algorithms that are executed prior to the fine-grained variability analysis to improve the results in situations with unclear relations between models (e.g., due to missing documentation).

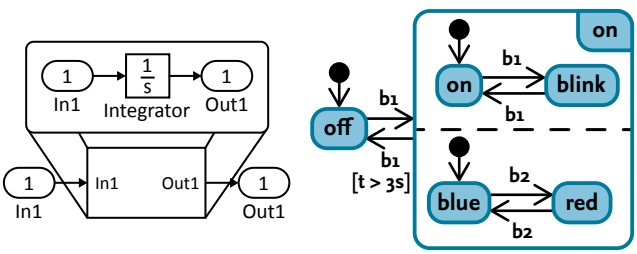
In Figure 4.4, we show the workflow that we derived from these observations. Starting with a set of model variants, we first execute our clustering step (cf. Section 4.4) to identify smaller clusters of models that are highly related and to identify and remove outliers. For each of the identified clusters, we execute the variability mining (cf. Section 4.5) to identify a so-called 150% model to store the identified variability (i.e., common and varying parts) for all compared models.

4.3 Background

In this section, we give background information on block-based languages (cf. Subsection 4.3.1) as well as identifying fine-grained variability information for related block-based models (cf. Subsection 4.3.2) [84, 192, 193, 194]. Our previous work on model clustering with SAMOS was already discussed in the previous chapters, so is omitted here.

4.3.1 Block-based Languages

A common means to develop solutions for complex problems in industry are *model-based languages* as they allow to describe domain knowledge and concrete implementations on an abstract level with reduced complexity [64]. The corresponding descriptions can be used to automatically generate executable code for different platforms or to perform model-based testing. *Block-based modeling languages* as a subgroup of these languages represent the functionality of software in the form of directed graphs. In most of these languages, the graph's *nodes* execute code written in a specific programming language (e.g., JAVA or C++) or represent *atomic functions* defined by the node's language (e.g., mathematical operations). Usually, the execution can be passed from one node to another by triggering *edges*. These edges often provide means to exchange data between the nodes and connect the *inports* and *outports* of nodes. These ports define the nodes' *interfaces*. A large part of block-based languages provides additional concepts to define abstractions from complex functionality by using *hierarchical nodes* encapsulating sub models with sub nodes defining the functionality on the corresponding hierarchy level. Depending on the used block-based language the terms used for the model elements (i.e., nodes, edges, ...) might differ.



(a) *MATLAB/Simulink* (b) Statechart

Figure 4.5: Example model instances for two block-based modeling languages

In Figure 4.5, we show two concrete model instances for block-based modeling languages. In Figure 4.5(a), we can see an exemplary *MATLAB/Simulink* model consisting of *blocks* linked with *connectors*. This model receives a data signal via the `In1` block, processes this data using the functionality specified in the *subsystem* block (i.e., a hierarchical node), and emits the resulting value via the `Out1` block. Using hierarchical nodes similar to the subsystem in this example, developers can encapsulate complex functionality into reusable elements (i.e., in this example the integrator block pipeline). By arbitrarily nesting such hierarchical nodes it is possible to develop complex systems on a more understandable level by refining abstract functionality with each added hierarchy level. In Figure 4.5(b), we present an exemplary statechart implementation for an LED with two colors (i.e., `blue` and `red`) consisting of system *states*. For each hierarchy level the execution start is unambiguously defined by the *initial state* (i.e., the black bullet). *Transitions* between the states allow to change the system state. For example, by pushing button

b1 the LED transitions from state off to the *parallel state* on. This state contains two *regions* (separated by a dashed line) that define the states of the LED's color and its current light mode. In contrast to *hierarchical states* these parallel states allow the execution to be in multiple states at once as they can have more than one region. For example, the LED can be in any combination of the states {on, blink} and {blue, red} depending on the inputs of the user via the buttons b1 and b2. In case the user pushes button b1 more than 3 seconds the LED is turned off again.

Looking at the presented examples, we can see that although both languages follow different purposes, they consist of nodes and edges linking them. However, they differ in the used paradigm (i.e., data-flow oriented language vs. state-oriented language). In addition, not all languages provide all discussed elements. While, for example, statecharts allow modeling of parallel executing using parallel states with multiple regions, *MATLAB/Simulink* only allows abstraction from complex functionality using subsystems. As a result, Wille et al. present guidelines to allow developers to correctly consider such differences during adaptation of their family mining for new languages [198].

4.3.2 Family Mining

In previous work, Wille et al. introduced *family mining* as a reverse-engineering technique to provide developers with fine-grained variability information between sets of block-based model variants [84, 192, 193, 194]. In Figure 4.6, we show its workflow consisting of the three phases *Compare*, *Match*, and *Merge*. Before executing the first phase, it divides the input models into a single *base model* (e.g., a smallest model) representing the basis for all comparisons and a set of *compare models* that are iteratively compared and merged with this base model using a pairwise algorithm. During the *Compare* phase it iterates through the models by analyzing the data-flow and identify possible variability relations between the model elements. Usually, the resulting set of comparison elements is ambiguous as for each model element multiple possible counterparts in the compared model exist. The *Match* phase analyzes this list of preliminary relations to identify for each element from a model at most one counterpart in the other model. The resulting list of distinct relations is then used in the *Merge* phase to merge the identified variability relations into a single 150% model. In case more compare models exist that were not yet processed by the algorithm, another iteration is started where the created 150% model serves as input for the base model. This way, it iteratively generates a 150% model of all compared models that can be visualized or further processed to generate an SPL [197].

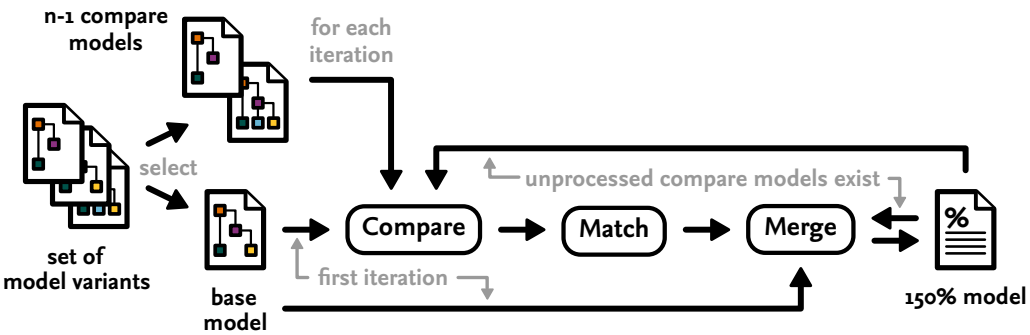


Figure 4.6: Iterative pairwise workflow for the family mining approach

4.4 Clustering for Variability Mining

We have used and extended our clustering technique to preprocess statechart variants for variability mining. For the overall workflow of clustering given in Figure 4.7, we detail each of the major steps in the following subsections. Note that the technique is completely language agnostic, in the sense that we can design metamodel-based extraction schemes to generate features for clustering a set of structural (typically graph-based) models conforming to such a metamodel. We have so far used the framework for other types of models including *Eclipse Modeling Framework (EMF)*, Unified Modeling Language (UML) and feature models with a corresponding *extractor plug-in* for each.

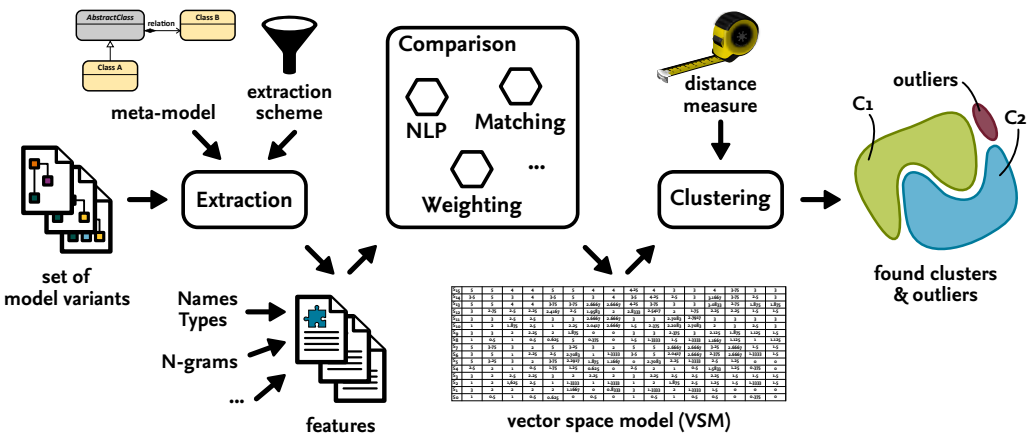


Figure 4.7: General overview of the clustering workflow

4.4.1 Extracting Features from Statecharts

The first step is to inspect the metamodel of the language (cf. Figure 4.9), and decide on the features to extract. Depending on the problem at hand, one can, e.g., choose to extract just the names of the model elements for a domain analysis scenario while ignoring the types of nodes or the graph-structure, or completely ignore the names and consider the types and graph structure for a clone detection scenario (specifically type-II clones [141]). Here we are interested to find variants close to each other in terms of element names, types and structure. The translation of this into our clustering framework is setting the feature type to *typed bigrams* (see Chapter 3).

Next, we inspect the metamodel of the language and design an extraction scheme. This is to be done by a domain expert who decides which parts of the metamodels are relevant for the problem at hand. Following the metamodel, we use the EMF API to code a simplified extraction scheme in JAVA including:

- Regions, States, Events: include their names and types.
- Transitions: include TransitionActions, Conditions along with their expressions/statements and types.
- Relations (to be encoded as bigrams):
  - Region –> State (containment),
  - State –> Event/TransitionAction/Condition (via transitions of State),
  - Event/TransitionAction/Condition –> State (via outgoingState of Transition).

To demonstrate, bigrams extracted from Figure 4.1(a) would include `[State,cls_unlock]-[Event,key_pos_lock]`, `[Event,key_pos_lock]-[State,cls_lock]` and so on.

4.4.2 Comparing the Features

Once the features are extracted, we decide on how to process and compare the features to build the *vector space model (VSM)*. Again, this is done by a domain expert who chooses the parameters of the framework considering the problem at hand. With reference to the details on parameters in the previous chapters, we have chosen to simplify the setting by turning off certain framework settings, e.g., weighting and advanced *natural language processing (NLP)*, such as using Wordnet for semantic relatedness. However, we do check the types, i.e., when comparing `[State-A]` with `[Region-A]`, the resulting similarity is set to 0.5 due to the type mismatch, despite the exactly same names (see relaxed type matching setting of SAMOS, as discussed in the previous chapters). See Table 4.2 for the configuration applied in this chapter.

The NLP features of the framework have also been extended to handle the names of the model elements in the statecharts. Inspecting the example statechart in Figure 4.1(a), one would immediately notice that names are typically given in snake case (`token1_token2_...`). We have used the built-in tokenization capabilities of the framework to process those names. We have further employed a simplifying pass to avoid parsing full expressions/statements in the conditions and actions for transitions: operators, boolean primitives, parentheses and brackets are given to the framework as stop-words, so that they are ignored when comparing names. The NLP overall allows the framework to detect similarities of event names (`key_pos_lock` vs. `key_pos_unlock`) or conditional expressions (`pw_enabled = true` vs. `pw_enabled = false`). Further NLP used includes Levenshtein similarity for typos, and stemming for cutting off the affixes.

4.4.3 Vector Space Model Computation and Clustering

The feature comparison scheme explained in the previous section is used to build the term frequency matrix for the VSM. The next step is to choose a distance measure and a clustering technique, which again depends on the problem at hand. We have used *Bray-Curtis* distance [66], as an approximate measure of the normalized (roughly the percentage) distance between two vectors. p and q being two vectors of n dimensions, *Bray-Curtis* distance is defined as:

$$bray(p,q) = \frac{\sum_{i=1}^{n} |p_i - q_i|}{\sum_{i=1}^{n} (p_i + q_i)}.$$

Note that while we certainly favor using normalized distances such as *Bray-Curtis* or normalized *Canberra* over absolute ones such as *Manhattan* (which roughly translates to graph edit distance). See 6 for a more detailed discussion on distance selection.

The *hierarchical agglomerative clustering (HAC)* algorithm with average linkage (i.e., inter-cluster distance equals the average pairwise distance of all the contained data points) is used on top of the vector distances to compute the dendrogram as the result. Inspecting the dendrogram, we detect outliers and cluster formations in the dataset. In our case study results, we show a sample dendrogram in Figure 4.11(a) on page 57. The numbers, i.e., the leaves of the dendrogram with labels `V*` represent data points as individual statechart model variants. The joints in the dendrogram correspond to a height (y axis with possible values in the range [0..1]), which is the normalized distance between two leaves or subtrees. A possible interpretation of the dendrogram would be that variants `V11611` and `V11616` are outliers (marked in red frames), with variants `V000*` forming a big cluster (marked in a blue frame). Note that how much further a

cluster can be decomposed into subclusters (e.g., into two subclusters in this case) depends on the interpretation.

4.5 Variability Mining for Block-based Languages

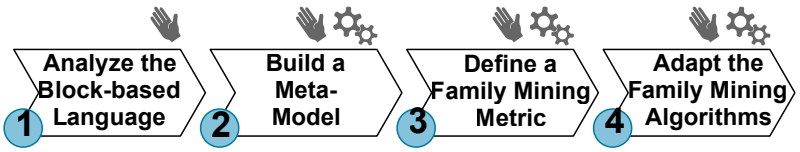


Figure 4.8: Guideline steps that are needed to adapt family mining for a new block-based language. *Hand* symbols highlight steps that have to be executed manually, while *gears* indicate steps that can be automated. Steps with both symbols can only be partially automated.

In this section we briefly sketch the variability mining algorithm by Wille et al., in contrast to the fully detailed description in our original publication [198]. Wille et al. propose guidelines to adapt their corresponding algorithms for block-based languages in four steps as depicted in Figure 4.8. The approach is not completely language-agnostic as it takes into account details about the used modeling language in the analysis. Thus, the first three steps are concerned with gaining knowledge that should be considered in the mining process and providing this knowledge in a form suitable to the algorithm. *Step 1* is dedicated to manually analyzing the used modeling language (cf. Subsection 4.5.1) to identify details that can be used to create a suitable metamodel representation of the language in *Step 2* (cf. Subsection 4.5.2). Such a representation builds an abstract view on the analyzed model elements and allows users to exchange the metamodel depending on the language analyzed during family mining. Besides manually building such a new metamodel, reusing existing metamodel representations of well-known languages is possible. Based on the gained insights and the user's domain knowledge a custom-tailored similarity metric can be defined in *Step 3* specifically describing the domain's perception of similarity for compared models and model elements (cf. Subsection 4.5.3). Afterwards the existing family mining algorithms (cf. Subsection 4.3.2) can be adapted in *Step 4*, allowing their execution for the analyzed language (cf. Subsection 4.5.4).

4.5.1 Analyze the Block-based Language

The foundation of the family mining approach is formed by model-based techniques to abstract from the concrete block-based language of the compared model variants. Thus, analyzing the language enables us to create such a metamodel representation for any new language. By identifying the right level of abstraction from the used language, we can create fitting metamodels to reduce the structures of models to the desired level of abstraction. This allows the family mining approach to focus only on details that are absolutely necessary to identify variability information for such models. Finding the right level of abstraction is a fine line between providing too little information for the algorithm to produce sensible results and considering too many details resulting in unnecessarily long execution times. Thus, profound domain knowledge of experts is needed to create a suitable metamodel for languages that should be analyzed with family mining (cf. Subsection 4.5.2). As a result, the detailed analysis of languages to gain such knowledge is considered as a completely manual step.

First of all, we need to either find existing metamodels for the block-based language if possible. An analysis of the language allow us to understand the language elements. We need this insight for selecting relevant language elements and properties to be used by the family mining algorithm. Language elements or corresponding properties that contribute to the overall functionality (e.g., blocks and their functions in *MATLAB/Simulink*) of model instances or allow to distinguish between them (e.g., their names) are regarded as relevant parts because they make the models comparable. Other language elements or properties that only represent *syntactic sugar* and can be transformed to an equivalent representation or do not contribute to the functionality at all are regarded as "irrelevant". Overall the relevant elements and their properties have to be selected carefully because otherwise created metamodels might not be expressive enough to model concrete implementations in the analyzed language. On the other hand, a metamodel storing too much information might result in long execution times for the family mining approach as too many details have to be processed.

4.5.2 Build a MetaModel

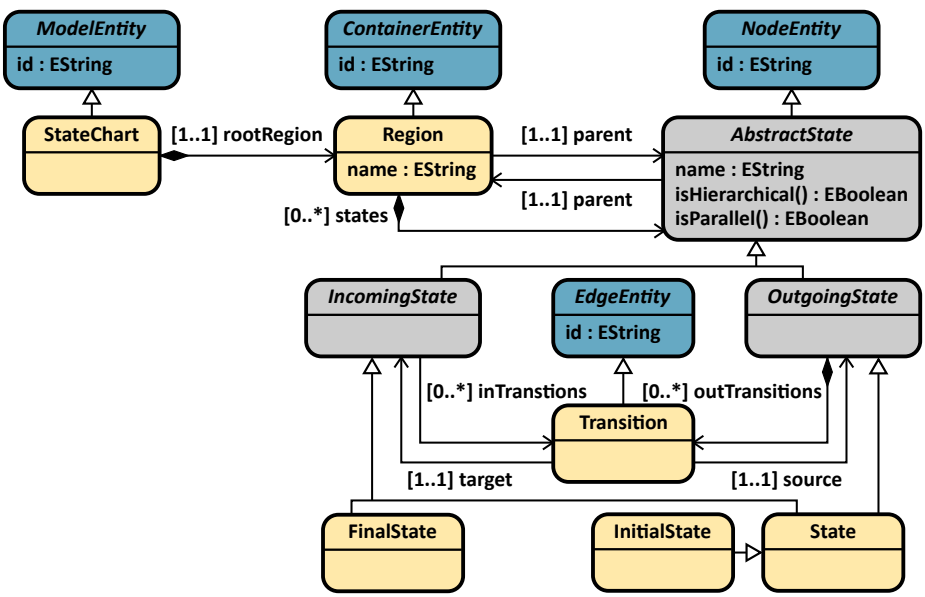


Figure 4.9: Excerpt from the metamodel for the family mining of statecharts

Based on the selected relevant model elements and corresponding properties it is now possible to build a metamodel or to modify an existing metamodel. The first step is to select a suitable metamodeling language such as EMF. Relevant model elements should be modeled using classes (e.g., `EClasses` in *Ecore*) with attributes (e.g., `EAttributes` in *Ecore*) modeling their relevant properties and inheritance applied where applicable (e.g., states are a generalization of initial states). In Figure 4.9, we show an excerpt from the metamodel for statechart family mining. The classes `ModelEntity`, `ContainerEntity`, `NodeEntity`, and `EdgeEntity` are part of the base metamodel. The created metamodel is able to express *semantically correct models* from the language if the users executed a *complete* language analysis with a *suitable* subset selection of relevant elements and realization of a *well-formed* metamodel (i.e., all elements and properties are modeled correctly) based on these results. To allow processing of the models by the family

mining algorithms, developers have to realize importers to create corresponding metamodel instances. In addition, suitable exporters have to be realized to transform the resulting annotated metamodel instances back to the original language notation.

4.5.3 Define a Family Mining Metric

The next step for the adaptation is to create a metric to allow comparison of elements and corresponding properties of metamodel instances. A sensible way to start is to execute a ranking of the selected element properties according to their influence on the overall functionality. Properties with a high impact (e.g., the function of a *MATLAB/Simulink* block) should be ranked higher than properties with less expressiveness (e.g., the name of a *MATLAB/Simulink* block). Based on the ranking, developers then can assign corresponding weights. These weights should be normalized (ideally in the interval $[0..1]$ as it allows direct translation to percentages). Thus, the summed up weights for the properties of an element should not be higher than the upper bound of the normalization interval (i.e., compared elements cannot have a similarity higher than 100%). For examples of concrete metrics, we refer to [193] and to [194] for metrics used in the comparison of *MATLAB/Simulink* models and statecharts, respectively. As a small illustration, the authors design a metric, where model similarity is defined in terms of 5% name, 75% function, 10% inport and 10% outport similarity.

4.5.4 Adapt the Family Mining Algorithms

After creating a metamodel representation for the analyzed language and defining a corresponding metric to allow comparison of corresponding metamodel instances, it is now possible to adapt the family mining algorithms. Next, we give a brief summary of the three family mining phases *Compare*, *Match*, and *Merge* (cf. Figure 4.6 in Subsection 4.3.2).

4.5.4.1 Compare Phase

During the *Compare* phase, the algorithm identifies possible relations between two compared models. For the first iteration, we have to select a *base model* that serves as a basis for the comparisons from the set of input models. All remaining models serve as *compare models* and are iteratively compared and merged with the selected base model. The technique allows for two approaches to select the base model. Either the user selects the base model manually based on domain knowledge or an automatic algorithm is used. In the current implementation the algorithm selects the smallest model as Wille et al. argue that extending an existing variant with additional functionality is a common clone-and-own approach. Thus, the smallest variant most likely represents the common core for the analyzed variants.

To compare the selected base model and one of the compare models, we iterate over the data-flow of the analyzed models. By starting at the highest hierarchy level, we start this data-flow analysis and compare the models' *entry points* that are defined by language-specific model elements (e.g., the initial states of the statecharts in Figure 4.1). The comparisons are continued by virtually separating the compared models into *stages* with each stage containing only elements that have the same *distance* (i.e., the number of nodes or edges) on their shortest path to the model's entry points.

Given that during the *Merge* phase (cf. Subsubsection 4.5.4.3) all variability is correctly merged into a 150% model, the generic family mining algorithms are able to walk through all alternative paths for model comparisons with $n > 2$ models. Thus, the compare algorithm can

identify possible relations between elements on these alternative paths and elements from the new compare model that is analyzed during the current iteration of the mining algorithm.

4.5.4.2 Match Phase

Usually, the resulting comparison elements are ambiguous as for each model element multiple possible counterparts in the compared model exist (e.g., for the transitions between state `cls_unlock` and state `cls_lock` in Figure 4.1 two possible matches exist). During the *Match* phase the list of all created comparison elements is processed to match each element from a model to at most one counterpart in the compared model. For each comparison element CE_i the algorithm first identifies all other comparison elements sharing either the base model element or the compare model element with CE_i. From the resulting list of comparison elements, it selects the element with the highest similarity value as a direct match and rule out all other possibilities by removing them from the list. This step is repeated until all model elements were matched.

In case no distinct match can be selected based on the similarity value (i.e., the comparison elements have the same similarity), all corresponding elements are sorted to the end of the comparison element list. This solution implicitly solves most conflicts by matching other comparison elements first. For cases where this strategy fails, a *decision wizard* is used which allows the user to either manually select the best match or apply additional user-defined automatic resolution strategies defined prior to the execution.

4.5.4.3 Merge Phase

After processing the matched list of comparison elements and establishing distinct relations between the compared models, the algorithm can merge a 150% model storing all identified variability relations. The first step for the merging is to categorize the relations identified for the model elements contained in each comparison element CE_i from the list. Here, it analyzes the calculated similarity and uses a function which maps similarity values ≥ 0.95 to mandatory, $(0, 0.95)$ to alternative and 0 to optional elements [84, 194].

Depending on the identified relations, the algorithm applies different strategies to merge the compared models into a copy of the base model. In case of *mandatory* elements, it marks the existing element in the base model as mandatory and annotate that it is contained in the base model *and* the compare model. In case of an *optional* element, it either annotates the already existing model element in the base model as optional or first merge the corresponding element from the compare model and afterwards annotate it. Finally in case of *alternative* model elements, it only merge the element into the 150% model that is not already contained in the base model copy. For comparisons with $n > 2$ models, it has to consider the already existing variability for all iterations $i > 1$ as here the 150% model from the previous iteration serves as base model (cf. Figure 4.6).

The resulting 150% model for the two `CLS` features from Figure 4.1 can be found in Figure 4.10. We can see for the three transitions that only one of them is contained in the `AutoPW` variant and the other two variants are contained in the `ManPW` variant. The transition from the `AutoPW` variant and the upper transition from `ManPW` variant are regarded as alternatives because the additional guard in the `ManPW` variant is their only difference. The remaining `ManPW` transition is regarded as an optional element. Both states and the transition from `cls_lock` to `cls_unlock` are regarded as mandatory. For clarity reasons, we did not annotate the concrete variability classes (i.e., mandatory, alternative and optional) in Figure 4.10 and neglected the source model annotations for mandatory elements.

The final 150% model can now be used to export the results to a graphical representation or for further processing by additional algorithms (e.g., to generate an SPL [197]). Furthermore

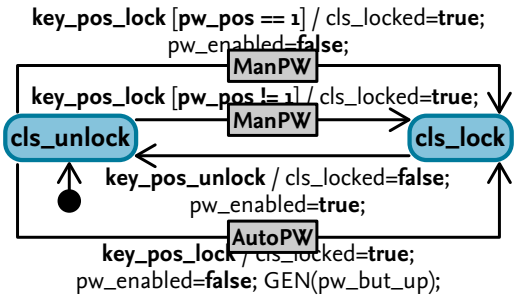


Figure 4.10: 150% model for the `ManPW` and `AutoPW` variants of the `CLS` feature in Figure 4.1

it serves as input for the next iteration of the family mining algorithms in case of $n > 2$ input models (cf. Figure 4.6).

4.6 Implementation

In this section, we give an overview of the implementations for our clustering framework (cf. Subsection 4.6.1) and the family mining framework (cf. Subsection 4.6.2). See Figure 4.4 for the overall architecture of the two techniques combined.

4.6.1 The Clustering Framework

The clustering framework is implemented partly in JAVA (feature extraction and comparison) and partly in R (clustering). While using *Eclipse* and EMF, the framework at the moment does not have an explicit *Eclipse* plug-in architecture; the components should be interpreted as conceptually distinct and modular steps of the workflow. Indeed for the study in this chapter, the default importer and extractor parts have been replaced with new ones for statecharts.

- *Language/Base MetaModels:* The *Ecore* metamodels to be used for loading the statechart models and extracting features.
- *Importer:* Standard importer for EMF resources.
- *Schemes & Parameters:* Specification of extraction, matching and comparison schemes and other framework parameters [22, 19].
- *Extract:* Feature extraction code, using the EMF API in JAVA. Here the extraction logic discussed in Subsection 4.4.1 is implemented.
- *Compare:* Feature comparison to build the VSM (cf. Sections 4.4.2 and 4.4.3). As we extract features into the internal representation of the framework, this component is used as-is with the appropriate parameter settings.
- *Cluster:* Distance calculation and clustering in R using packages `hclust` for *hierarchical agglomerative clustering (HAC)* and `vegan` for Bray-Curtis distance (cf. Section 4.4.3)
- *Visualization:* Export of the cluster hierarchy plot for visual identification of clusters and outliers.

4.6.2 The Family Mining Framework

The family mining framework is implemented using JAVA and the *Eclipse* plug-in mechanisms to allow for easy extension with customized algorithms.

- *Family Mining Core:* The framework's core plug-in provides all basic classes for user interaction (e.g., to configure or trigger algorithms for selected files) and the execution of the family mining workflow. Developers can customize this workflow by integrating additional plug-ins via the provided extension points for the following framework plug-ins.
- *Importer & Exporter:* These plug-ins allow to import models from block-based modeling languages files (e.g., *MATLAB/Simulink* model files) to the internal metamodel representation or to export the results (e.g., in form of reports or the original modeling language). As the import and export of models to/from an internal metamodel involves model-to-model transformations, realization of such plug-ins cannot be automated. However, we provide basic import and export plug-ins for *Ecore* files.
- *Base MetaModel:* The base metamodel plug-in contains artifacts that can be used to create a language-specific metamodel for our generic mining algorithms (cf. Subsection 4.5.2). It is realized using EMF *Ecore*.
- *Language MetaModel:* This metamodel is needed to have an internal representation of imported models. Its generation can be automated after a manual analysis of the language (cf. Subsection 4.5.1). It is realized using EMF. Its connection to the base metamodel is optional (cf. Subsection 4.5.2).
- *Compare:* The compare algorithm is part of the core plug-ins and uses the generic structure specified by metamodels (cf. Subsubsection 4.5.4.1).
- *Metric:* Metric plug-ins can be automatically generated after analyzing new languages and specifying corresponding weights (cf. Subsection 4.5.3).
- *Match:* The match algorithm is completely language-agnostic (cf. Subsubsection 4.5.4.2) and, thus, part of the core plug-ins.
- *Decision Wizard:* Decision wizards can be automatically generated with an extended *domain specific language (DSL)* description or metamodel annotations (cf. Subsubsection 4.5.4.2).
- *Merge:* Merge algorithms have to be manually implemented as block-based languages are too diverse for generating such algorithms (cf. Subsubsection 4.5.4.3).

4.7 Case Study

We combined our clustering technique in Section 4.4 with the family mining in Section 4.5 to execute fine-grained variability mining on clusters of related models excluding outliers. We used the workflow described in Figure 4.4 and focused on the question how our cluster and outlier detection can improve such results:

- *RQ4.1 – Outlier Detection:* Is the clustering technique capable of eliminating outliers in input models before executing the family mining approach?
- *RQ4.2 – Cluster Detection:* Is the clustering technique capable of identifying sensible clusters of related models in the input models?
- *RQ4.3 – Improvement of Results:* Are the results of the family mining improved by applying it only to identified clusters and neglecting outliers?

4.7.1 Case Study Subjects

For our case study, we selected different subjects to evaluate whether the described clustering is capable of improving the mining results (cf. Subsubsection 4.7.1.2).

	Variants			Shared	Mutually Exclusive	Alternating
	Cluster 1	*Cluster 2**	∑			
OD1	8	2	10	7	14	6
OD2	8	2	10	10	12	5
OD3	8	2	10	6	0	21
CD1	8	8	16	17	2	8
CD2	12	12	24	12	7	8

*containing the *outliers* for the OD scenarios and the *second cluster* for the CD scenarios

Table 4.1: High-level overview of the five scenarios from the *body comfort system (BCS)* used to evaluate the cluster detection (CD1 & CD2) and outlier detection (OD1 – OD3).

4.7.1.1 Family Mining Implementation

We selected the block-based language *statecharts* for the evaluation of our approach. For that language, our partners already realized manual implementations using the guidelines from previous work without exploiting their common structures and implementing the algorithms for each language separately [195]. We use the same subjects as for the previous evaluation by our partners [197, 194]. Thus, for the evaluation of the statechart family mining, we concentrate on the *body comfort system (BCS)* also used as a motivating example in this chapter (cf. Section 4.2). The *BCS* implementation represents a real world system from the automotive domain that was realized using *IBM Rational Rhapsody* statecharts and was decomposed into an SPL [109, 128]. The resulting SPL comprises 27 reusable features and allows generation of 11,616 valid variants. These features encapsulate the functionality of different system parts (e.g., the central locking system or alarm system – cf. Section 4.2). Depending on the feature selection, the *BCS* statechart variants comprise up to 70 states, 40 regions and 94 transitions. In particular, we concentrate on 17 *BCS* variants that were derived from the *BCS* SPL to cover a wide range of functionality in the feature combinations [109, 128].

4.7.1.2 Cluster and Outlier Detection

For the evaluation of our cluster and outlier detection, we use five scenarios in the context of the *BCS* SPL (cf. Section 4.2 and Subsubsection 4.7.1.1) that we outline in Table 4.1: To evaluate the capability of the outlier detection algorithms to identify a small set of outliers in a set of otherwise highly related variants (cf. *RQ4.1*), we selected three outlier detection scenarios (OD1 – OD3). These outlier variants differ compared to the cluster of related variants as large parts of their selected features differ. Apart from the outlier detection, these scenarios also evaluate that the algorithms are capable of identifying the expected cluster of related variants. Furthermore, to evaluate the capability of the clustering algorithms without outliers present (cf. *RQ4.2*), we selected two cluster detection scenarios (CD1 & CD2) with two delimitable clusters of variants. Each selected scenario comprises clusters of valid feature selections from the *BCS* SPL to generate corresponding variants. Analyzing the selected scenarios, we also evaluate the benefit of the outlier and cluster detection (cf. *RQ4.3*).

In Table 4.1, we show the number of variants contained in the different scenarios. Each scenario consists of two clusters. While *Cluster 1* always represents one of the expected clusters, *Cluster 2* either represents the set of outliers for the OD scenarios or the expected second cluster for the CD scenarios. The table shows, for each scenario, the number of shared features (i.e., contained in *both* clusters), the number of mutually exclusive features (i.e., contained only in *one* of the clusters) and the number of alternating features (i.e., features that cannot clearly

Component		Setting	Description
Extraction	model	statechart	extractor for statechart models
	unit	type-name pairs	further units NA at this point
	structure	bigrams	capturing structure
	postprocessing	off	no postprocessing
Comparison	basic NLP	on	compound-word similarity w/ internal NLP
	advanced NLP	off	WordNet disabled
	type matching	relaxed	0.5 multiplier for non-exact types
	struct. matching	semi-rlx	semi-relaxed comparison for n-grams
VSM	frequency	sum	sum of frequencies
	idf	off	idf disabled
	weighting	off	weighting disabled
Analysis	distance	Bray-Curtis	normalized distance
	clustering	hclust	hierarchical clustering with average linkage
	cut	manual	manual extraction of clusters

Table 4.2: SAMOS configuration for the case studies.

be assigned to variants from a particular cluster). Thus, we evaluate our clustering technique using variants with different degrees of similarity (i.e., number of shared or mutually exclusive features). In addition, we evaluate the resistance against "noise" (i.e., alternating features) which might negatively influence the clustering as variants might be assigned to unexpected clusters. Using the selected scenarios for the cluster and outlier detection provides us with a ground truth as we clearly labeled variants as outliers or which cluster they belong to. As a consequence, we can evaluate whether the detection is capable of generating results confirming this ground truth and, thus, meets the expectations of experts well familiar with the *BCS* implementation.

For consistency, we also list the table of settings of SAMOS (Table 4.2 for the case studies in this chapter.

4.7.2 Methodology

For our evaluation, we execute our proposed approach for each scenario selected in Table 4.1, and discuss the resulting clusters and outliers, as well as the 150% models generated for these clusters.

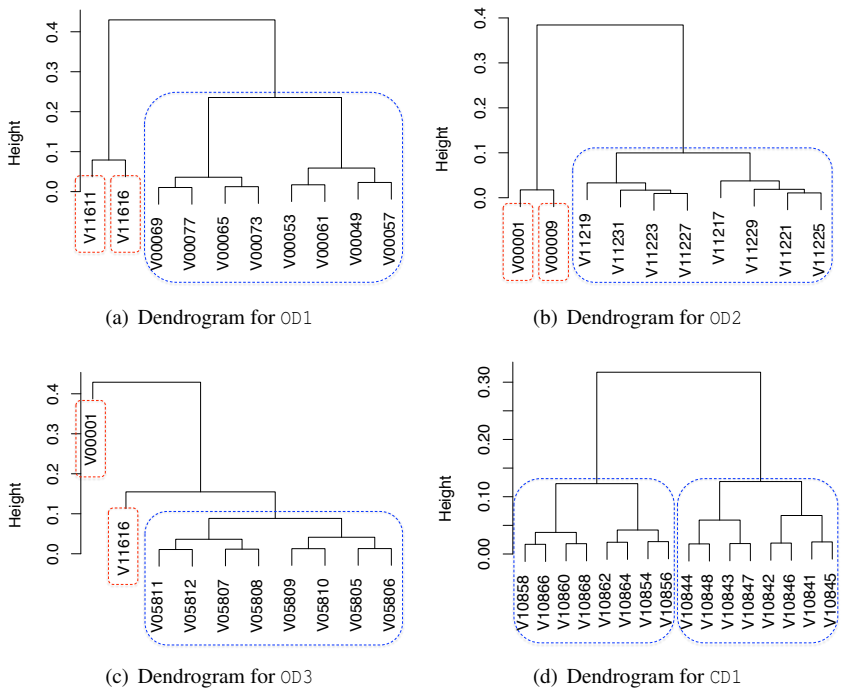
4.7.3 Results and Discussion

In this section, we report on our results of our case study and discuss them with respect to our research questions.

RQ4.1 – Outlier Detection We applied our clustering technique with the settings outlined in Section 4.4 for each scenario. By inspecting the dendrograms generated by the clustering, we identified the outliers in scenarios `OD[1-3]`. Figures 4.11(a) to 4.11(c) depict the corresponding dendrograms. The interpretation of the figures is as follows: (1) a big coherent cluster of data points with high similarity marked in a blue frame; and (2) individual data points with relatively little similarities with the main cluster marked in red frames.

An important point to discuss is related to the distinction between a feature vs. its implementation in the variant models. While we designed the scenarios based on selecting/deselecting

features to comprise a notion of similarity between models, we ignored their implementation, especially how big their corresponding realizations in the models are. This may lead to some situations, e.g., having a common feature with a very large implementation offsets the selection of all other minor features and dominates our similarity calculation. This is partly reflected in Figure 4.11(c), where a considerable number of different features (6 out of 27) between `V11616` and the large cluster contribute only to around 10% difference. Elaborate weighting schemes, e.g., based on importance of features or model elements is omitted considering the scope of this work.



(a) Dendrogram for `OD1`

(b) Dendrogram for `OD2`

(c) Dendrogram for `OD3`

(d) Dendrogram for `CD1`

Figure 4.11: Dendrograms for the cluster detection scenario `CD1` and the outlier detection scenarios `OD1` to `OD3`. Clusters are marked with blue frames and outliers with red frames.

RQ4.2 – Cluster Detection For cluster detection we adopted an approach similar to the outlier detection; namely we inspected the resulting dendrogram and this time tried to find large groups of similar models rather than isolated outliers. The dendrograms for `CD1` and `CD2` are largely similar; thus we only depict the former for the sake of space. Looking at the dendrogram in Figure 4.11(d), it can be seen clearly that there are two distinct sets of data points, with high similarity among each other (around 0.10), yet the intra-cluster distance is around 0.30; high enough to comprise separate groups. As mentioned previously, especially in the case where a much larger number of models is considered, it is arguable whether to obtain a few but large clusters, or many but small (sub-)clusters; it is up to the domain expert to make this design decision.

The results of the outlier and cluster detection scenarios confirm that our clustering technique is able to perform well enough with respect to the expectations of the experts building up the

ground truth for this study. This holds also in the presence of alternating features explained previously as being potentially challenging. While the number of models considered in the case study is limited, considering the results we answer *RQ4.1* and *RQ4.2* positively.

RQ4.3 – Improvement of Results To examine the results of the family mining algorithms with or without a cluster and outlier detection, we distinguish two situations: a) cases *with* outliers and b) cases *without* outliers.

In case outliers exist, we identified that our outlier detection improves the fine-grained variability information generated by the family mining algorithm. The main reason is that outlier variants represent models that have a *low* or at worst *no* relationship to the remaining input models. Executing our family mining without detecting these outliers might result in unexpected variability relations in the 150% model or even elements that have no relation to the rest of the 150% model. Thus, we argue that using the outlier detection is essential in situations where users are not fully familiar with the input models and relations between these models are unclear (e.g., after developers left the company).

In case no outliers exist, we do not necessarily need to execute our cluster detection as the generated results for the complete set of input models represent valid variability information. However, executing family mining on particular clusters might allow users to focus their analysis on the corresponding models. Furthermore, detecting clusters reduces the chance of unexpected variability information (e.g., induced by variants from other clusters; cf. Subsection 4.2.2). Thus, detecting clusters prior to executing the fine-grained variability mining can improve the experience of users. However, we think that these clusters should at least be evaluated at a high level by users as, depending on the focus of users, it might be interesting to combine multiple clusters for a bigger picture of the overall system. For example, this could be interesting for senior developers working on multiple projects that are identified as separate clusters.

Overall, we answer *RQ4.3* positively as using the cluster and outlier detection allows to improve the experience of users. The generated 150% models are tailored towards their expectations as outliers are removed and the identified variability information can be focused on particular clusters.

4.7.4 Threats to Validity

Although we designed, implemented and evaluated our technique with great care, different threats to validity are inherently present. Our feasibility studies contain models from the automotive domain only and the used case studies are limited to *IBM Rational Rhapsody* statecharts. This limits the generalizability of our approach as we can only say with certainty that our algorithms and generic implementations work for these particular case studies and languages. However, we used the generic family mining and the outlier and cluster detection algorithms without having a particular domain in mind and prior to our evaluation. Thus, we kept ourselves from being biased and are confident that the algorithms are applicable to other block-based languages and models from other domains.

On the clustering part of this work, there are also several threats of validity. First of all, the technique itself aims to deliver a fast but approximate overview of the given data; hence it may not be ideal if very high accuracy is required. The scenarios considered in this chapter are relatively simple and further scenarios with varying size and complexity, preferably in real industrial settings, could be investigated to test our claims. Another point, already addressed above, is that the ground truth, hence the expectation of the domain experts, is shaped with respect to the high level features in contrast to the lower level implementation in terms of the actual models. This contrasts with the clustering technique working on the level of implementation, i.e., the model

variants. Elaborate weighting schemes based on features and/or model elements could be introduced to mitigate this situation, where the domain experts associate the corresponding parts with varying importance to guide the clustering.

4.8 Related Work

In this section, we discuss related work for our presented clustering algorithm (cf. Subsection 4.8.1) and our variability mining (cf. Subsection 4.8.2).

4.8.1 Clustering Techniques

The existing approaches for model comparison typically consist of expensive pairwise techniques that focus on accurate comparison/matching of just two models [159]; *EMFCompare* being one of the most used industry standards. Only in recent work [19, 22, 35], scalable techniques based on information retrieval and clustering (i.e., machine learning) for comparing large number of models have been introduced. To the best knowledge of the authors, there is no further related work in the *model-driven engineering (MDE)* domain with a comparable perspective, i.e., a holistic and scalable treatment of models for analysis and visualization.

In the SPL domain, Zhang et al. [204] use model comparison (i.e., *EMFCompare*) to synthesize a product line model from variant models. Our approach is different in the sense that the mining/merging technique does not use model comparison directly, but rather clustering is meant as an individual step of data preprocessing, selection and filtering before family mining. Furthermore, model clustering is faster and more scalable than pairwise techniques [22, 35], though possibly with a trade-off in accuracy. This trade-off fits our workflow perfectly where we need a rough overview of a potentially large number of models, whereas family mining operates by itself in a precise way in the next step.

4.8.2 Variability Mining Techniques

Comparing different types of models for various purposes (e.g., variability identification or model versioning) has been extensively investigated [159] and can be categorized into *clone detection*, *differencing* and *variability identification*.

Clone Detection Different clone detection algorithms exist for various software artifacts and languages including models [103, 141, 159]. For example, graph-based algorithms allow to detect syntactic clones [64, 108, 132], semantic clones [7] and near-miss clones (i.e., clones that have minor differences) [132]. Further algorithms translate models to textual representations for text-based clone detection [10]. Clustering identified clones Alalfi et al. identify varying parts in *MATLAB/Simulink* models by inferring that all remaining parts represent variability [8]. While the approach can theoretically be adapted for other languages, its applicability has only been demonstrated for *MATLAB/Simulink* and no corresponding guidance exists. In contrast the family mining framework by Wille et al. provides a generic implementation and adaptation guidelines for adapting further languages.

Differencing For a complete analysis of variability information in models, focusing on the cloned parts only is not sufficient as the differences describe their variability. A large number of differencing algorithms exist for various languages and software artifacts in literature (e.g., [52, 95, 96, 186, 201]) and in commercial tools (e.g., *DiffPlug* [68], *SimDiff* [73]). However, unlike

the family mining approach these algorithms only provide information on the differences and neglect the common parts necessary to describe the models' complete variability.

Variability Identification While clone detection and differencing only identify one side of the variability (i.e., either the common *or* the varying parts), variability identification combines both dimensions. In addition, classic clone and difference detection algorithms mostly compare only two models, while for a complete variability analysis in a set of models all models need to be compared.

An essential part of variability identification is merging the compared models and storing the identified information in a unified representation. A large number of model merging algorithms exist for various contexts (e.g., [11, 117, 152, 181, 189]). These algorithms solely focus on merging the information of the models and neglect their variability. In contrast, additional variability-aware algorithms exist that merge the compared models and annotate the elements' source models [77, 144, 147] or visualize the identified variability [115]. Unlike the familiy mining algorithm these lack explicit variability information (i.e., about mandatory, alternative or optional parts) and, thus, are not applicable for fine-grained variability analysis.

Nejati et al. describe an approach that similar to family mining uses heuristics (e.g., metrics) for comparing and matching elements from statechart variants [123, 124]. However, in contrast to the family mining approach, Nejati et al. only merge model elements with annotations about their parent models and neglect fine-grained variability information, which limits their solution to the generation of contained variants. In contrast, the family mining approach not only allows transition to an SPL allowing generation of these variants [197], but also detailed analysis of the fine-grained variability. Another variability identification approach similar to family mining is that of Ryssel et al. [149]. However, unlike family mining they do not focus on storing the identified variability in a unified representation but focus on extracting reusable library elements [151]. Work by Font et al. focuses on identifying variability information of models [76] and incorporating the developers' domain knowledge to identify larger reusable model parts similar to Ryssel et al. [75]. While these approaches store the identified artifacts in an SPL using the *Common Variability Language (CVL)*, Wille et al. focus on generating 150% models that can be translated to different SPL representations (e.g., delta-oriented SPLs [197]). Klatt et al. focus in their work on identifying variability between related source code artifacts [97]. While, similar to family mining, they operate on a graph-based representation, their algorithm relies on *abstract syntax trees (ASTs)* of the analyzed code and, thus, is limited to the underlying data-structure. In contrast, Wille et al. 's presented model-based approach is applicable to different block-based languages and they also showed that similar algorithms can be applied to source code [196].

The most part of mining techniques in literature use so-called *pairwise* approaches. In contrast, *n-way* algorithms are capable of merging a potentially arbitrary number of model variants at the same time. In literature, such algorithms exist to merge different artifacts such as *UML* models into 150% models [147] and model-transformation rules into variability-based rules [172]. While the family mining approach can be easily adapted for different languages with varying paradigms (i.e., *MATLAB/Simulink* and state charts), these approaches were evaluated for single languages and currently lack such capabilities.

While all these approaches concentrate on the concrete realization artifacts, other techniques exist to extract high-level configuration options in form of feature models [145] or CVL models. Examples are approaches that analyze natural-language requirements [188], product maps [150, 158], or existing products [204]. During an SPL generation from the identified 150% models (e.g., [197]) such information could provide a configuration model for the generated SPL.

4.9 Conclusion and Future Work

In this chapter, we explained in detail how fine-grained variability information can be identified for large sets of models in different block-based languages using the family mining approach supported by our technique presented in this thesis. We demonstrated and discussed how our language-agnostic cluster and outlier detection can improve the variability information generated by our family mining. Using the presented extension it is now possible to remove outliers (e.g., completely unrelated variants) from a set of input models and cluster them into more meaningful sets (e.g., relevant for particular users). Such fine-grained variability mining could in turn support model management in SPL-like scenarios.

In future work, we plan to implement a direct link between the family mining framework and our clustering framework. Currently, these two frameworks are realized independently and after executing the cluster and outlier detection the user has to manually interpret the dendrograms. Using this information outliers have to be manually removed and sensible clusters have to be selected prior to the detailed variability analysis using the family mining. By applying additional algorithms to cut the generated dendrogram trees at sensible positions, we plan to (semi-)automate this step by at least automatically providing suitable suggestions to the user. In addition, we plan to evaluate clustering techniques that can help to automatically select a base model for the comparisons using other (possibly more accurate) heuristics than selecting the smallest model.

Chapter 5

Managing a Feature Model Repository

Model-Driven Engineering and Software Product Lines promote the use of models as central artifacts for a variety of activities including domain analysis and generative software development. As these paradigms gain popularity, the number and variety of models in use increase. Several initiatives to gather models in repositories exist, such as ATL Zoo for metamodels or S.P.L.O.T. for feature models, aiming for public access and reuse. However, as those repositories are only partly or not at all curated, the growing number of models leads to problems such as duplicates a.k.a. clones, and lack of repository overview. This makes both repository management and model searching/reuse very hard. We address this issue for S.P.L.O.T. by adapting SAMOS, our generic model analytics framework, for feature model comparison. We perform two exploratory case studies. First, we aim for getting a high level repository overview with large clusters and their domains. Secondly, we try to get clusters of highly similar models, to be interpreted as duplicates or clones. We conclude our approach is applicable for feature models and can improve the use and maintenance of S.P.L.O.T.

5.1 Introduction

Model-Driven Engineering (MDE) and Software Product Lines (SPLs) are paradigms heavily using models for a variety of activities ranging from domain analysis to software development, deployment and testing. While one of the key objectives of such paradigms is the management and reuse of increasingly complex software artifacts, the same problem emerges as they gain popularity and wider adoption: there are more, larger and more complex models in use [26]. Recently, there has been some effort to collect various models in model repositories to facilitate public access and reuse. Notable examples are the ATL Ecore Metamodel Zoo[1], OCL dataset along with the metamodels [121] and Software Product Lines Online Tools (S.P.L.O.T.) feature model repository[2] [119]. One problem of such repositories is when they are either partly or not at all curated.

[1] http://web.emn.fr/x-info/atlanmod/index.php?title=Ecore

[2] http://www.splot-research.org/

This is particularly evident in S.P.L.O.T.: a quick inspection of the individual models reveals that (1) models usually lack proper metadata on their domains, versions, etc.; (2) there are quite many duplicates/clones/versions of models with no explicit relationship noted. Moreover the number of models in the repository increases rapidly, scaling up the aforementioned problems. These have serious implications in scenarios involving both repository management and use. First of all, there is a lack of repository overview, e.g. what groups of models there are, and to which domains these belong. This type of information would enable repository exploration, facilitating model search and reuse. Secondly, as new models are added, either the model manager or the users themselves are burdened with the manual labeling of the models e.g. with respect to their domains. And lastly, there is a considerable amount of duplicate models, clones arbitrarily copy-pasted, and also various versions of the same models lying around in the repository.

These issues have been raised in the domain of MDE [22, 35]. A promising solution is the automatic comparison of models [159] for gaining some information on the repository dataset such as grouping/subgrouping of models, proximities among models (and groups as well) and outliers. Doing this on a large scale for hundreds of models requires techniques beyond the complex and expensive pairwise comparison such as in [118]; rather it requires approximate but fast and scalable techniques. These include e.g. fragmentation of models into smaller chunks, typically via Information Retrieval (IR)-based and statistical methods such as clustering [22, 35], especially for clone detection [18].

There has been a considerable amount of work in the SPL community on feature model analysis, comparison and use of IR-based techniques, however with several important distinctions. First of all, inspecting the thorough literature study of Benavides et al. [37] on automated analysis of feature models reveals that analysis is mostly performed on a single feature model and some configuration of that, for instance to find out the dead features or valid products. Other approaches involve multiple feature models as input, but model comparison is generally perceived based on the configuration semantics (as used by She et al. [158] in contrast with ontological semantics): feature models are transformed into logical formulas, and reasoned about their pairwise relationships such as generalization/specialization [178], or exact differences [2, 48]. Another approach uses EMF Compare to calculate pairwise differences between feature models [70]. An interesting take on feature model comparison is presented by Xing [200], who argues that feature models might evolve over time with changes in both the structure and feature names/descriptions, and applies their generic model differencing technique to feature models using the structural (or ontological according to [158]) information in the models. On the other hand, many researchers have proposed IR and clustering, not for comparing feature models but requirements, product descriptions, or features themselves (e.g. their names, the text in their description) for reverse engineering feature models [14, 158]. Along a line of work mostly on model synthesis and composition [3, 4], Bécan et al. utilize IR and NLP techniques in their interactive model synthesis tool [36]. To our best knowledge, there has been no comparable work in the modelling and SPL domains to cluster large numbers of feature models with our objectives of discovering relations among large datasets in a scalable manner.

In this chapter, we attempt to apply our generic model analytics framework to compare the feature models in the S.P.L.O.T. repository. Our goals are twofold; introducing our approach to the SPL community which we believe can benefit from the proposed techniques, and testing the genericness and extensibility of our approach for new model types and datasets. First, we extend our framework with an extraction scheme for feature models using the S.P.L.O.T. Java API for parsing Simple XML Feature Model (SXFM) files. Using many utilities of the framework, notably Natural Language Processing (NLP) tools, we test our approach on the 1034-model dataset in S.P.L.O.T. We perform two case studies: firstly trying to get relatively large sized clusters and their corresponding domains in the repository; and secondly obtaining clusters of very similar

models—to be interpreted as duplicates, clones or versions. We conclude our approach is indeed applicable for feature models and can improve the use and maintenance of S.P.L.O.T.

5.2 Analyzing Feature Models

In this section we start with some preliminaries and move on to detail our approach for analyzing and comparing feature models by extending SAMOS.

5.2.1 SXFM Feature Models

There are many notations for feature models, starting with the original one by Kang et al. [93], later extended with cardinalities, additional constraints, attributes and so on [156, 157]. As a starting point for this study we take the SXFM notation supported by the models in S.P.L.O.T. A feature model has a feature tree with different types of features in it (*Root* as the root of the tree, *Solitaire* meaning singleton, non-grouped features), optional/mandatory modifier, feature groups with cardinalities (lower and upper bounds) and grouped features in them, and the parent-child relations. They may also contain additional constraints in Conjunctive Normal Form (CNF) clauses. See Figure 5.1 for an example SXFM feature model with mandatory, optional and grouped features.

5.2.2 Extracting n-grams and Constraints

We want to extract the information from the features (names, types, cardinalities) and their relation to other features in the hierarchy (i.e. structural context) in the form of n-grams as supported by our framework. Additionally, we want to represent constraints for accurate comparison. Using the SXFM Java parser library of S.P.L.O.T., it is rather straightforward to traverse the feature tree and generate the information to be used for clustering. We present here the pseudocode for a simple extraction of bigrams and constraint sets from feature models. Note that the bigram representation is an inaccurate simplification for grouped features (due to the then implementation of SAMOS) as the n-ary relation among the grouped feature and its members together is transformed into binary relations. Please see Section 5.4 on future work to overcome this limitation with more complex features, i.e. trees.

We give an example extraction from a model in S.P.L.O.T. (Figure 5.1) for $n = 1,2$ and constraints in Table 5.1. We use a mobile phone feature model, with mandatory features (*Calls*), optional ones (*GPS*), and an alternative feature group (meaning only one should be chosen) with *Basic*, *Colour* or *High Resolution* screen. Constraints not being shown in the figure would include ($\sim$ *_r_2 or* $\sim$ *_r_3_5_6*), which describes the implication *GPS* $\rightarrow \sim$ *Basic* when we map the feature id's to the corresponding feature names.

5.2.3 Rest of the SAMOS Workflow

Once we obtain the IR-features, the rest of the framework can be used as is for the n-grams. For the constraint sets, we apply the Hungarian algorithm [105] to obtain a best (partial) match score among the sets based on their feature names and negations (each using vertex similarity in SAMOS). In terms of vertex/node matching, i.e. how to compare unigrams with each other, users can choose to check for synonyms via tokenization, filtering, stemming/lemmatization, Levenshtein distance and WordNet[3]; whether types should be exactly the same or ignored altogether.

[3] `https://wordnet.princeton.edu/`

Algorithm 2 Processing the feature models for bigrams and constraints.

declare list $featureList$

procedure $process(FM)$
  $processTree(T)$
  $processConstraints(Cons, T)$

procedure $processTree(T)$
  parent ← extract $(name, type)$ pair from current node T
  for each regular child C of T **do**
    $child$ ← extract $(name, type)$ pair from child C
    if type(C) == mandatory **then**
      $edge$ ← "child[1..1]"
    else
      $edge$ ← "child[0..1]"
    end if
    $featureList \leftarrow featureList \bigcup (parent, edge, child)$
    $processTree(C)$
  end for
  for each grouped feature G of T **do**
    $child$ ← extract $(name, type)$ pair from child G
    $edge$ ← "child[$i..j$]" where i, j are lower/upper group cardinality
    $featureList \leftarrow featureList \bigcup (parent, edge, child)$
    $processTree(G)$
  end for

procedure $processConstraints(Cons, T)$
  declare set S
  for each CNF constraint F in $Cons$ **do**
    for each term Trm in F **do**
      $name$ ← get name of Trm in feature tree T
      if negated(Trm) **then**
        $name \leftarrow \sim name$
      end if
      $S \leftarrow S \bigcup name$
    end for
    $featureList \leftarrow featureList \bigcup S$
  end for

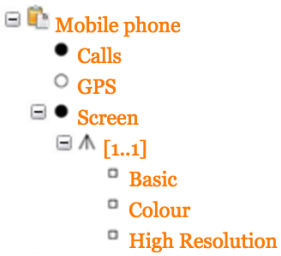


Figure 5.1: A feature model example (constraints not shown).

type	IR-features
unigram	(Root-Mobile phone) (Mandatory-Calls) (Optional-GPS) (Mandatory-Screen) (Grouped-Basic) ...
bigram	(Root-Mobile phone)-(child[1..1])-(Solitaire-Calls) (Root-Mobile phone)-(child[0..1]-(Solitaire-GPS) ... (Solitaire-Screen)-(child[1..1])-(Grouped-Basic) (Solitaire-Screen)-(child[1..1])-(Grouped-Colour) ...
constr	(∼GPS,∼Basic) (High resolution,∼Basic) ...

Table 5.1: IR-feature extraction: some examples for Figure 5.1. '∼' denotes negation.

Finally, users can choose to apply type-based weighting (e.g. some parts of the model might be more important such as classes vs. parameters in UML) and idf.

Having set all the above schemes, the framework computes the VSM based on the n-grams extracted. Using this matrix and picking a distance measure (e.g. cosine for domain clustering), clustering is performed in R. Further options are what type of clustering to do (flat vs. hierarchical) and algorithm-specific parameters. The main output of hierarchical clustering is the dendrogram, which can be manually inspected, or *cut* with certain parameters to automatically infer clusters (e.g. for clone detection with threshold values).

5.3 Case Studies

We performed two exploratory case studies to demonstrate the applicability of our approach for feature models, on the 1034-model dataset in S.P.L.O.T. (as of July 18, 2018[4]).

[4]snapshot available at `http://www.win.tue.nl/~obabur/data/AMMORE18.zip`

Component		Setting	Description
Extraction	model	feature model	extractor for feature models
	unit	name	only names
	structure	unigrams	ignoring structure
	postprocessing	on	token expansion, filtering
Comparison	basic NLP	on	single-word similarity w/ basic NLP
	advanced NLP	on	WordNet semantic relatedness (Lin)
	type matching	off	NA for names
	struct. matching	off	NA for names
VSM	frequency	sum	sum of frequencies
	idf	norm. log	normalized log
	weighting	off	type-based weighting NA for names
Analysis	distance	cosine	angular cosine distance
	clustering	hclust	hierarchical clustering with average linkage
	cut	semi-automatic	filtering cut + manual inspection

Table 5.2: SAMOS configuration for case study 1.

5.3.1 Case Study 1 - Repository Overview and Major Domains

In this case study, we want to obtain large groups of related feature models, to be able to identify roughly the domains in the repository (e.g. mobile phone models). Observing that in our case the domain knowledge is captured mostly in the feature names, unigrams ($n = 1$) are adequate here. We have adopted a similar parameter set previously used for clustering the ATL Ecore metamodels [22] (see Chapter 2): unigrams of names only (no types), NLP including pre-tokenization for compound words, Levenshtein distance for typos, stemming, lemmatization and WordNet for semantic relatedness; normalized log idf weighting, cosine distance and finally hierarchical clustering with average linkage (see Table 5.2) The procedure for this case study is as follows: (1) cluster the whole dataset with the above settings, (2) perform a filtering pass to cut off the models that are less similar (≥ 0.8 cosine distance, arbitrarily chosen as *high enough* similarity) to the rest of the dataset, and focus on relatively large clusters (≥ 20 models), and (3) perform a second clustering step on the subset and visualize the dendrogram. See Table 5.2 for a summary of the configuration of SAMOS for case study 1.

Note that the filtering step is necessary, as we have to manually inspect and evaluate the results; manually handling a 1034-item dendrogram with a non-trivial coarse structure within the scope of this work is not feasible. Figure 5.2 is useful to see the diversity of the models in S.P.L.O.T: there is not much thick branching, for instance dividing the dataset into few large clusters.

A Brief Qualitative Evaluation The filtering steps reduced the dataset size to 275. The resulting dendrogram for clustering those 275 models is given in Figure 5.3. The interpretation of the dendrogram is that (1) the numbers on the dendrogram correspond to the table row indices of the feature models as given in S.P.L.O.T. table and (2) the joining height of individual branches are the normalized distance (can be considered percentage dissimilarity) between those two individual models or groups of models. Cutting the dendrogram horizontally at height 0.8, we obtain 10 major clusters, as shown in Figure 5.3 in dashed lined boxes with cluster labels at the bottom. Inspecting the models, we can roughly attribute the following domain labels to the clusters: cluster 1 of mobile media and cluster 2 of mobile phone models, cluster 3 of models with many

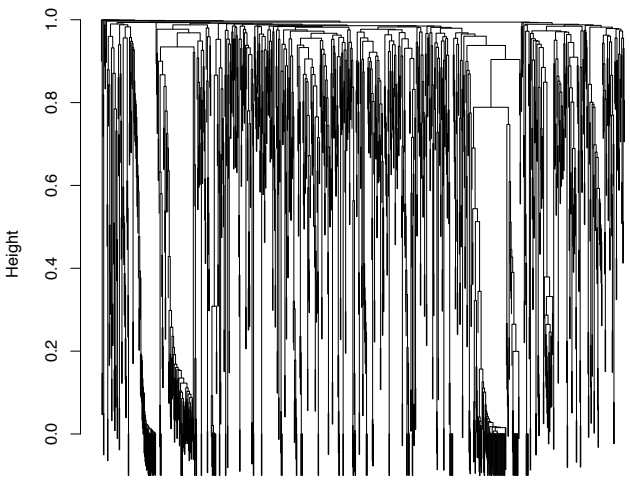


Figure 5.2: The dendrogram of the 1034 models. Leaves are hung from the joints (which denote the actual similarity) for better visualization; hence some leaves extend below 0.

feature names *Feature-1, Feature-2, ... Feature-N*, cluster 4 of voting/e-voting models in Portuguese (urna), cluster 5 of models with many feature names *F1, F2, F_1*, and so on; cluster 6 of models in Spanish about real estate (inmobiliaria), cluster 7 of models for marketplaces (mostly in Spanish), cluster 8 of models with a lot of abbreviations and numbers as feature names (see discussion below), cluster 9 of e-shop and e-commerce models, cluster 10 of computer models (mostly in Spanish).

A precise account of the accuracy of this categorization is difficult to give, as the dataset itself is not labeled. Instead we comment about the clusters and some false positives we found by manual inspection. Clusters 1 and 2 are very accurate, and we could find cases where our tool successfully detected typos and NLP-related changes. Cluster 3 has the upper part of the branch (say, higher than 0.5 distance) seemingly less and less relevant; models with few percentages of features with names *Feature-X* are detected as partly relevant to this cluster.

Cluster 4 is also a quite accurate account including non-English (in this case Portuguese) models, although our tool cannot specifically address them at the moment (e.g. with synonym detection in other languages). Cluster 5 ($F's$) is conceptually similar to 3, but does not have as many irrelevant models and is mostly accurate. Clusters 6 and 7 are again mostly accurate except for an outlier numbered 682 (about mass transport). Cluster 8 is the most irrelevant one, where a lot of different models with abbreviations and numbers as feature names are grouped together. Clusters 9 and 10 also seem quite accurate. In the latter, we even identified two models about computers in different languages being correctly clustered together thanks to some shared terms.

A detailed account of the recall for this case study is left as future work. One can increase the recall (at the cost of precision) by relaxing the parameters/thresholds more generously. Note that there are certainly some more domains in the dataset to be discovered, e.g. car and bike feature models. If there are too many domains, it might be practical to handle all of the dataset manually; a semi-automatic way could solve the problem and is left as future work.

Figure 5.3: The dendrogram depicting 10 domain clusters.

Component		Setting	Description
Extraction	model	feature model	extractor for feature models
	unit	attributed	key-value pairs for attributes
	structure	edged bigrams, sets	capture tree structure and constraints
	postprocessing	off	lemma/case standardization
Comparison	basic NLP	on	compound-word sim. w/ internal NLP
	advanced NLP	off	WordNet disabled
	type matching	relaxed	0.5 multiplier for non-exact types
	struct. matching	semi-rlx, Hung	semi-rlx for bigrams, Hungarian for sets
VSM	frequency	sum	sum of frequencies
	idf	off	idf disabled
	weighting	off	weighting disabled
Analysis	distance	msk. Bray-Curtis	masked normalized distance
	clustering	dbscan	density-based clustering
	cut	automatic	automatic clustering with thresholds

Table 5.3: SAMOS configuration for case study 2.

5.3.2 Case Study 2 - Detecting Duplicates and Clones

In this case study, we set the objective to obtain groups of very similar feature models, both content and structure-wise. We would like to detect duplicates, clones, variants and versions in S.P.L.O.T. , easily seen with a brief inspection of the models in the repository. Exact categorization of the found models into one of these is beyond the scope of this work; we refer to all of those simply as clones. As we want to capture as much information as possible, we turned to use full bigrams (with types, cardinalities) and constraints here. We used (1) no idf weighting, (2) relaxed type matching (with non-exact type matches getting a reduced similarity multiplier). We further used masked Bray-Curtis Distance with a density-based clustering technique (please refer to Chapter 6 for full details for this technique and [103] for clone detection in general). In summary, we ran SAMOS with the clone detection setting on the 1034-model dataset for detecting Type A, B and C clones [18] with respective distance thresholds of 0 (identical except cosmetic changes), 0.1 (slightly different) and 0.3 (somewhat different). See Table 5.2 for a summary of SAMOS configuration for case study 2.

clone type	#clusters	#pairs	#models involved
A	22	64	59
B	60	1472	208
C	90	3320	382

Table 5.4: Clone detection statistics.

A Brief Qualitative Evaluation We found a considerable number of clones; we report in Table 5.4 the number of clone clusters, the total number of clone pairs and models involved in those.[5] This already indicates that a relatively high percentage of the models in S.P.L.O.T. is highly similar to other models in the repository. The actual clone clusters with some examples

[5] See `http://www.win.tue.nl/~obabur/data/AMMORE18.zip` for the full list of clones

type	model indices in a cluster
A	811 814
A	763 769 773
A	165 187
A	567 568 571 573 589
A	11 24 26 31 34 54
A	...
B	1029 1030
B	967 970
B	884 892 919 927
B	599 622 629
B	479 602
B	...
C	1032 1034
C	783 794
C	556 735 835
C	11 24 26 28 31 34 37 47 54 61 65 67 72 77 241 374
C	19 50 98 125
C	...

Table 5.5: Some of the clone clusters.

are given in Table 5.5. Inspecting a (random sample amounting to %20 of) the clone clusters, we were able to trace the following:

- Type A clones: SAMOS was able to detect Type A clones (implying no significant change) very accurately; we found no incorrect labeling in the validation subset. Manually inspecting the clusters, we found the following changes which led to a Type A classification: change in the date of creation of the model, feature model name, metadata, constraint names (i.e. not the content), order of elements in the feature tree and the CNF formulas, consistently changed feature id's (which lead to e.g. completely different looking constraint formulas), and cosmetic changes in feature names (e.g. upper/lower and snake/-camel casing).

- Type B clones: SAMOS detected clones with a variety of changes, ranging from simple modification of cardinalities, textual changes in feature names (e.g. typos, additional tokens) to addition or removal of features and constraints, and moving of features to elsewhere in the feature tree as well. Although all the clones we inspected were relevant, some might arguably be categorized as higher level, namely type C (see discussion about weighting and feature groups in Section 5.4).

- Type C clones: SAMOS is again in most cases accurate in finding higher percentage of addition, removal, or changes in feature trees and constraints, although we identify certain shortcomings. SAMOS treats feature names such as *Feature-1* and *Feature-2* as highly similar, which leads to an inaccurate Type C classification (see e.g. line 3 of type C clones in Table 5.5). Also due to the simplification of grouped features in the form of bigrams, SAMOS is not able to distinguish very well between e.g. grouped features in a strict alternative feature group of cardinality 1..1, and the same features moved outside as mandatory

features (again with cardinalities 1..1). The problem with weighting as mentioned above might lead to some misjudgment and is subject to improvement.

5.4 Discussion and Future Work

The case studies show our approach can provide an insight into the feature models in S.P.L.O.T., in terms of repository overview and domain decomposition, and of duplicates a.k.a. clones. We consider these as hypotheses for further verification in our future studies. We extract the information captured in the feature names, the ontological hierarchy of the feature tree and the constraints (syntax only); and use this to efficiently calculate approximate similarities among models. Here we provide no quantitative evaluation on the accuracy of our approach partly due to the lack of a labeled dataset and the exploratory nature of the study. A brief qualitative evaluation yet reveals our approach is indeed effective to a considerable extent with room for improvements. It would be interesting to quantitatively evaluate the effect of different settings, given a labeled dataset (e.g. feature models with explicit domains for the first case study, or mutated feature models for clone detection). In this section we discuss several limitations of and improvements for our approach.

Grouped features, configuration semantics: Given our choice of bigram representation, we inaccurately extract group cardinalities for each grouped feature. An improvement would be to switch to tree representations in SAMOS (which was still in development at the time, see Chapter 6 for these integrated into SAMOS) to properly capture those. Furthermore, we do not perform any inference and compare the syntactic constructs as is. Hence, it should be further investigated how we can incorporate the inferred information, though then the approach would move towards comparing knowledge bases. Another further step would be incorporating attributes in extended feature models for comparison, which is not supported by the S.P.L.O.T. dataset, and left as an open problem.

Weighting, fine tuning: At the moment we did not use any weighting scheme in this chapter, such as type-based weighting (e.g. constraints having less weight than the feature tree) as supported by SAMOS, but also advanced ones. An initial idea for the latter would be depth-dependent weighting, i.e. features lower in the tree hierarchy get lower weights, hence attributing more importance to higher level features (which are arguably more general or abstract, e.g. as mere structural units, more coarse grained/architecture-related). Inspecting the results of the clone detection, we believe a fine-tuned weighting scheme could improve the clone classification, especially around boundaries between Type B, C and higher thresholds.

NLP settings: The S.P.L.O.T. dataset brings several new challenges for our framework's NLP capabilities, notably due to its multi-language heterogeneous nature. There are models in English, Spanish, Portuguese, Indonesian, etc. in the repository, which renders our English-based NLP tools inadequate. In an orthogonal direction for improvement, the framework could be extended with multi-language NLP, including e.g. tokenizers and even cross-language synonym checkers. The features labeled as *Feature-1* or *F1*, or cryptically abbreviated, pose yet another challenge.

Threats to validity There are several threats to validity for our work, mostly stemming from the exploratory approach. The settings we have chosen for the case studies may not be the most

efficient and accurate ones, but were chosen mostly for simplicity and demonstration purposes. A quantitative evaluation of different parameters and thresholds, and more importantly on labelled datasets (e.g. explicit domain labels for case study 1, clones for case study 2) would be required for a more precise account.

5.5 Conclusion

In this chapter we have presented an application of our generic model clustering technique to feature models. We have extended our framework to extract information from feature models and efficiently but approximately compare models. With two exploratory case studies on the 1034-model dataset in the S.P.L.O.T. repository, we get (1) a repository overview and major domains therein, (2) very similar models in the repository such as duplicates and clones. Based on the studies, we conclude that our approach is indeed applicable for clustering feature models. Following the two objectives we set in the beginning, we both confirm the genericity and applicability of our approach for different types of models, and provide a new perspective on the comparison of feature models for the SPL community. Indeed, our approach can help with the use and maintenance of emerging repositories such as S.P.L.O.T.; hence serves those aspects of model management in general.

Chapter 6

Metamodel Clone Detection with SAMOS

Wider adoption of model-driven engineering leads to an abundance of models and metamodels in academic and industrial practice. One of the key techniques for the management and maintenance of such artifacts is model clone detection, where highly similar (meta-)models and (meta-)model fragments are mined from a typically large set of data. In this chapter we have extended the SAMOS framework (Statistical Analysis of MOdelS) to clone detection, exemplified on Ecore metamodels. The clone detection uses and extends the framework's feature extraction, vector space model, natural language processing and clustering capabilities. We performed three extensive case studies to demonstrate its accuracy both quantitatively and qualitatively. We first compared the sensitivity and accuracy of SAMOS for metamodel changes through mutation and scenario analysis (which simulate clones) with those of NICAD-Ecore and MACH, tools for clone detection on Ecore and UML models respectively. We then compared the precision and recall of SAMOS and of NICAD-Ecore on a real dataset, consisting of conference management metamodels from the ATL Zoo. Finally we performed a repository-wide mining of metamodel clones from GitHub. We conclude that SAMOS stands out with its higher accuracy and yet considerable scalability for further large-scale clone detection and other empirical studies on metamodels and domain specific languages.

6.1 Introduction

Model-driven engineering (MDE) promotes the use of models (and metamodels to which they conform) as central artifacts in the software development process[1]. This eases development and maintenance of software artifacts (including source code generated from models), yet increasing MDE adoption leads to an abundance of models in use. Some examples of this include the academic efforts to gather models in repositories, and large-scale MDE practices in the industry [26, 120, 187]. This leads to challenges in the management and maintenance of those artifacts. One of those challenges is the identification of model clones, which can be defined in the most general sense as duplicate or highly similar models and model fragments [64]. Similar

[1] In our context, we refer to metamodels and models shortly as *models*, as metamodels are models too.

scenarios apply in the traditional software development for source code clones. There is a significant volume of research on code clones, elaborating the drawbacks of having clones, which can be a major source of defects or lead to higher maintenance cost and less reusability, and providing detection techniques and tools [143]. Note that in some cases clones might be useful too, as argued by Kapser et al. [94]; it is nevertheless worthwhile to investigate them. Code clones have attracted the attention of the source code analysis community, who had to deal with the maintenance of large numbers of artifacts for a longer time than the MDE community.

Model clone detection, on the other hand, is a relatively new topic. Many researchers have drawn parallels from code clones, and claimed that a lot of the issues there can be directly translated into the world of models. While the problem domains are similar, the solution proves to be a challenge. Source code clone detection usually works on linear text or an abstract syntax tree of the code while models are in general graphs [65]; other aspects are also inherently different for models, such as tool-specific representations, internal identifiers, and abstract vs. concrete syntaxes [167].

There are several approaches for model clone detection, or model comparison in the broader sense, in the literature [65, 159]; yet we are particularly interested in ones with a publicly available tool to be reused in our studies. A good portion of such tools are either limited to, tailored for, or evaluated on specific types of models such as MATLAB/Simulink. Notable examples, along with the code clone detector back-ends for them, are CloneDetective-ConQAT [64] and the SIMONE Simulink clone detector [9]—referred to as NICAD-SIMONE in this chapter—an extension of text-based tool NICAD to cover clone detection in Simulink models. Another related approach for Simulink models is ModelCD based on graph comparison and approximation [132], but the tool is not publicly available. While the approach is similar to ours in this chapter, given the unavailability we cannot use it in our study; see however Section 6.7 for a discussion. Störrle presents an approach and a tool, MQ_{lone}, for UML model clone detection [167, 168]. Störrle elaborately describes and classifies UML model clones, noting the differences to code clones and Simulink model clones. Furthermore, the author reports a much higher performance and scalability for MQ_{lone} compared to ConQAT and ModelCD. MQ_{lone} is integrated into the publicly available tool suite MACH[2], though in a very limited manner, with almost no control over the rich set of algorithms and settings developed by Störrle.

In our research we have the goal of detecting clones in large repositories of models and large evolving industrial domain-specific language (DSL) ecosystems based on the Eclipse Modelling Framework (EMF). Metamodels are artifacts of particular interest to us, for various purposes including metamodel repository management and DSL analysis (see Section 6.2 for a detailed discussion). To achieve this goal, we have investigated the feasibility of existing tools, with three major requirements: (1) conceptual and technological applicability to Ecore metamodel clones; (2) sensitivity to all possible metamodel changes, and accuracy in general (precision, recall); and (3) scalability for large datasets. As a starting point we considered MACH and NICAD-SIMONE as promising candidates. However, these tools underperformed with respect to some of our requirements, which will be demonstrated in the rest of this chapter.

We have taken an orthogonal approach by extending the SAMOS framework (Statistical Analysis of MOdelS) for model clone detection. SAMOS is a state-of-the-art tool for large-scale analysis of models [17], as introduced in the previous chapters. We wish to exploit the underlying capabilities of the framework—incorporating information retrieval-based fragmentation, natural language processing, and statistical algorithms—for model clone detection. In this chapter, we describe how we have extended and tailored SAMOS for (meta-)model clone detection.

The rest of the chapter is structured as follows. Section 6.2 opens with scenarios and con-

[2] https://www.pst.ifi.lmu.de/~stoerrle/tools/mach.html

ceptualization of metamodel clones. We outline the two clone detector tools used to contrast SAMOS to in our case studies, namely NICAD-ECORE and MACH, in Section 6.3. In Section 6.4, we elaborate on the extended SAMOS framework with its feature extraction, comparison, natural language processing and clustering capabilities. Section 6.5 details the three extensive case studies with mutation/scenario analysis for SAMOS, NICAD-Ecore and MACH; comparison of SAMOS with NICAD-Ecore on ATL Zoo metamodels; and finally a repository mining scenario on a very large set (thousands) of GitHub metamodels. The rest follows with an overall discussion including future work in Section 6.6 and further related work in Section 6.7. We finally draw conclusions on the applicability of SAMOS to metamodel clone detection.

6.2 Metamodel Clones

The goal of our research is to detect metamodel clones. Metamodel clones might exist due to a wide range of reasons including copy-paste or clone-and-own approaches in model-driven development [65], lack of abstraction mechanisms in metamodels for language design [174], or difficulty in reuse for DSLs in general [57]. Maintenance, which has been identified by Kosar et al. [102] as one of the major overlooked areas in DSL research, is hampered by the presence of clones and can benefit from clone detection [175].

In our research, we are interested in finding similar (fragments of) metamodels, with the following problems at hand. First and foremost, clones suggest potential scenarios for quality assurance and refactoring in MDE/DSL ecosystems. Also, as those ecosystems in large-scale settings do consist of multiple DSLs, clone detection across different DSLs and their versions forms a basis of empirical studies on their development and evolution. Furthermore, repositories and datasets of metamodels, whether company-wide in industry, online in the public domain, or specific collections for research purposes, could benefit from clone detection for activities such as repository management, exploration, data preprocessing, filtering, and large-scale empirical studies (such as on the origin, distribution and genealogy of the clones). Finally, we might use clone detection for plagiarism detection and assistance in grading student assignments for language design and metamodelling courses, similarly as done by the counterparts in the source code domain, such as JPlag [134].

While metamodel clones have not been specifically studied in the literature, model clones have been, particularly model clones in MATLAB/Simulink and other data flow type models. A very large portion of the model clone detection literature is focused on data flow languages such as Simulink with the following classification scheme [9, 7]:

- *Type-I* (exact) model clones: Identical model fragments except for variations in visual presentation, layout and formatting.
- *Type-II* (blind renamed, or consistently renamed) model clones: Structurally identical model fragments except for variations in labels, values, types, visual presentation, layout and formatting.
- *Type-III* (near-miss) clones: Model fragments with further modifications, such as changes in position or connection with respect to other model fragments and small additions or removals of blocks or lines in addition to variations in labels, values, types, visual presentation, layout and formatting.
- *Type-IV* (semantic) clones: Model fragments with different structure but equivalent or similar behaviour.

Note that we omit Type-IV, i.e. semantic, clones for the scope of this chapter as they pose an orthogonal and arguably bigger challenge; however this is perfectly in line with the related work for NICAD-SIMONE and MACH, both of which omit semantic clones as well.

While the above scheme is more or less a community standard, metamodels are more similar to UML class diagrams than to Simulink models, with respect to the following two key aspects: (1) the importance of the model element names, and (2) the dominance of a containment tree structure. Hence our conceptualization and classification of metamodel clones is mostly adopted from UML clones [168]. Störrle emphasizes that names of model elements are essential parts of UML models. His classification for UML clones has a notable distinction from the Simulink classification: it rules out Type-II (renamed) clones due to the indispensability of element names. This is also the case in the context of EMF metamodel clones. Our clone classification, adding a few items related to Natural Language Processing (NLP) to Störrle's classification, is given below. He further argues that the structure of UML models is dominated by a containment tree with few additional cross-tree connections. We believe those two observations apply to Ecore metamodels as well; though an empirical study to find evidence on large corpora is left as future work. Note we use a different formulation than of Störrle for semantic clones. However, we omit Type-D semantic clones for the scope of this chapter (see discussion above for Type-IV clones), which is consistent with Störrle who avoids semantic clones as well. We leave it as future work to conceptualize and detect semantic model clones. We will use the following classification for the rest of the chapter.

- *Type-A* duplicate model fragments except secondary notation (layout, formatting), internal identifiers.
  - Plus *any cosmetic change* in the names (lower-/uppercase, snake-/camel-case and other insignificant changes).
- *Type-B* duplicate model fragments with *small percentage of* changes to names, types, attributes, few additions/removals of parts.
  - Plus *potentially many syntactic/semantic changes* in the names such as typos, synonyms, semantically related words.
- *Type-C* duplicate model fragments with *substantial percentage of* changes/ additions/removals of names, types, attributes and parts.
- *Type-D* semantically equivalent or similar model fragments with different structure and content.

Now that we have set the concepts and classification of metamodel clones, we continue with existing model clone detectors that can be used for metamodel clone detection.

6.3 Other Model Clone Detector Tools

In this section we discuss two prominent model clone detector tools we used in our comparative evaluation against SAMOS.

6.3.1 NICAD-SIMONE

The NICAD Clone Detector[3] is a scalable clone detection tool implementing the NICAD (Automated Detection of Near-Miss Intentional Clones) code clone detection method [59]. It was mainly designed for finding intentionally copy/pasted units, such as functions and subsystems, that have been modified. It uses a configuration file to specify steps such as normalization of identifier names and filtering of irrelevant parts. It has good reporting capabilities in XML and HTML for readability. NICAD supports a range of languages and normalizations, and is designed to be easily extensible using a component-based plug-in architecture. Furthermore, it is scalable to very large systems with millions of lines of code.

Internally, NICAD is a parser-based and language-specific but lightweight tool, which adopts a line-based textual comparison rather than subtree comparison to achieve better performance and scalability. It is built on top of the TXL (short for Turing eXtender Language) programming and transformation language [58] for identifying syntactic clones while relying on pretty-printing to eliminate formatting differences and noise. Thanks to TXL, the tool is extensible to other languages via the introduction of the appropriate TXL grammar and transformation rules [142].

For the scope of this chapter, the main feature of interest is the standard mode of operation in the model clone detection tools derived from NICAD (notably SIMONE [9]). The approach consists of (1) parsing textual forms of models which may use various formatting and preprinting, (2) pre-processing to a normalized format, removing irrelevant parts, and ordering textual elements using multi-attribute topological sorting, (3) extracting subparts with the selected granularity for the comparisons such as subsystem scope for Simulink, (4) post-processing to further normalize, for instance rename identifier names, filter or transform elements, (5) line-based computation of least common subsequence (LCS) to find clone pairs, (6) clustering the pairs using connected component analysis.

There are two points to discuss in detail about this approach. For the LCS algorithm, the order of the elements naturally matters. Step (2) is needed to properly normalize reordering of elements because different orderings of the same elements still represent clones such as with data-independent declaration statements in code [140] or certain elements in models (UML models [167], Simulink models [9], metamodels [18]). For instance, NICAD-SIMONE indeed underlines a fundamental problem of line-based comparison on graphical data: the order of (at least some of the) elements actually does not lead to any meaningful change. NICAD-SIMONE tackles this by a sorting-normalization pass where it sorts model elements according to their type, name and so on, and therefore can successfully detect clones with reordered elements. Another point is the line-based comparison in LCS, which is done in NICAD outside the TXL scope (in a Turing+ script) and therefore hard to do in a language-aware manner. In our experience this line-based approach is followed consistently in NICAD-based approaches, whether for model clone detection as by Antony et al. for UML sequence diagrams [16] or code clone detection[4].

6.3.1.1 Extending NICAD for Metamodel Clone Detection

At Eindhoven University of Technology, we conducted a small project for an Ecore extension for metamodel clone detection. The base extension can be found in GitHub[5]. It is inspired by NICAD-SIMONE, i.e. the extension of NICAD for Simulink models, and tries to replicate the same mode of operation for detecting Ecore metamodel clones. It has the following major steps for its workflow:

[3] `https://www.txl.ca/nicaddownload.html`

[4] See a small Java code exercise in our report `http://www.win.tue.nl/~obabur/publications/JVLC18/`

[5] `https://github.com/jzhang3/Nicad3-emfatic`

1. Convert Ecore metamodels into the textual Emfatic[6] format,
2. Provide an Emfatic TXL grammar for NICAD,
3. Provide an Ecore plug-in with transformation rules for extraction, filtering, renaming and sorting of model elements.

In order to do a correct clone detection and fair comparison with other tools, we have developed an improved extension of this plug-in (replicating the mode of operation of NICAD-SIMONE as close as possible) with the following features:

- *Granularity.* Allow extraction with two fixed scopes: whole-model, or EClass (a metaclass of Ecore, similar to Class in UML class diagrams),
- *Filtering.* Remove most of the original filtering and simplifications (as implemented by our student), to keep as much information as possible for completeness, still excluding elements which SAMOS ignores such as EAnnotations (a metaclass of Ecore for annotations, see Section 6.4.2 for details),
- *Sorting.* Implement multi-attribute topological sorting wherever applicable (such as among all elements contained in an EClass, multiple supertypes and so on) with respect to type, eType and name of the corresponding elements,
- *Configuration.* Add configuration files to detect Type A, B and C clones with different threshold values; respectively 0, 0.1 and 0.3.

6.3.2 MACH

The MACH toolset for UML model analysis and checking includes the UML model clone detector MQ_{lone} developed by Störrle [167, 168]. The approach is applicable for multiple types of UML models, ranging from class diagrams to use case and sequence diagrams; in contrast to several existing approaches focusing on a single type of model, such as MATLAB/Simulink models [132]. The most relevant steps of the approach from the perspective of this chapter are (1) processing (XMI-serializations of) models as graphs, and encoding the graphs as Prolog programs representing node and edge information; (2) defining various similarity heuristics based on element names or hash/index values; and finally (3) using Prolog rules and inference to match, rank, and weight clones in the models. MACH natively supports UML files produced by tools such as MagicDraw; and automatically eliminates/normalizes layout, internal identifiers and tool-specific information.

The author advocates focusing on node similarity rather than the graph structure similarity (as done in other tools such as [64, 118]); and claims that UML models are very wide and flat trees with mostly containment structure in contrast to more general graphs. The clone detection is based on a variety of heuristics. First, name-based similarity heuristics involve comparing element names based on the Levenshtein edit distance and taking into account other factors such as different format, for example CamelCaps vs. separate words. Such heuristics include one which takes just the name of a single node for comparison, and another which also considers the neighboring nodes' names for similarity. In [168], a new heuristic is introduced where type and attribute information is also taken into account. Another set of index-based heuristics involve comparing the hash values of model elements: for instance, using a hash function which adds up

[6] https://www.eclipse.org/emfatic/

all characters found in two model elements. Finally, there is a more powerful similarity heuristic which can weight and rank clones, and hence favors more promising matches as clone candidates.

An important note is that the rich set of heuristics and settings developed by Störrle is only partially accessible in the closed source MACH toolset. The single standard clone detection setting in MACH calculates similarities with absolute measures, rather than standardized in the range of [0,1] or in percentages — which makes it harder to interpret the similarity. Furthermore, MACH finds clone pairs but does not group clones into larger clusters.

6.4 Using and Extending SAMOS for Clone Detection

As introduced in the previous chapters, the framework SAMOS is a state-of-the-art tool for large-scale analysis of models. In this section we elaborate how we have used and extended it for clone detection, highlighting the differences particularly with respect to the domain analysis settings in Chapters 2 and 3. Please refer to those chapters for the background, basic workflow and an example-driven operation of SAMOS.

6.4.1 Scoping/granularity for extraction

While, originally, SAMOS handles entire models and extracts all the model elements contained, we introduce the notion of scoping to define the granularity of independent data elements. For Ecore metamodels, we define three scopes: the whole model, EPackage, and EClass; it is relatively straightforward to pick these fixed scopes as the major *hubs* in the containment tree. The scope guides the extraction by mapping a model into one (i.e. whole model) or more data points such as per EPackage contained. Given a fixed scope, we adopt the approach for UML clone detection [168]: we cover all the model elements under transitive containment closure of that starting model element.

6.4.2 Extracting model element information

The main unit of information extracted by SAMOS has previously been the so called *type-name* pair, which essentially maps to a vertex in the underlying graph of the model. Such a pair encodes the domain-specific type (metaclass) information, such as EClass, and the name, such as Book, of a model element. For proper clone detection, we need to cover the attributes in the model elements (for instance whether an EClass is abstract) and cardinalities, e.g. of EReferences. We also need to explicitly capture the edges such as containment to include in the comparison. Therefore, we have extended the original feature hierarchy in SAMOS (see Figure 6.1) with (1) *AttributedNode*, which holds all the information of a vertex including domain-specific type, name, type, attributes as key-value pairs; and (2) *SimpleType*, which indicates whether an edge is of type *containment* or *supertype* (meaning superclasses as named in EMF). All the mentioned feature types belong to the *SimpleFeature* class and represent non-composite stand-alone features.

There are several implementation details worth mentioning about the extraction:

- Our current implementation covers almost the full Ecore meta-metamodel for extraction, except EAnnotations, EFactories, and generic types. We also ignore the attributes of namespace URIs and prefixes. The reason for exclusion for all the elements mentioned was our initial assessment of those (based on anecdotal evidence of our preliminary studies and discussions with language engineers) as mostly insignificant model content.

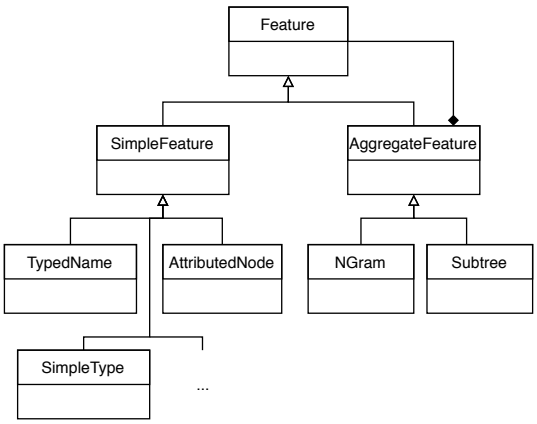


Figure 6.1: Feature hierarchy in SAMOS: simple features for representing vertices and edges, and aggregate features for encoding structure.

- Observing that for most attributes in Ecore metamodels, a mode or default value exists, we encode only non-default values in our AttributedNodes to reduce the data size and speed up the comparison. The fixed values can be deduced either from Ecore metamodel itself (default value), or by inspecting the majority of cases in our dataset (mode value).
- We push the type (eType) information (e.g. what EDataType is assigned to an EAttribute) into the AttributedNode itself, rather than representing it as a separate vertex connected to that node, which would be expected from a naive/plain graph-based extraction. Examples of the modified extraction can be seen in the vertices v_2 and v_6 in the upcoming example. It avoids the problem we observed with the settings of [23] (Chapter 3), in which our approach matched too many irrelevant features just because of matching types, for instance all EAttributes with the type EString.
- The framework can also be configured to do a normalizing pass on element names to convert them into a standardized corresponding lower and snake case, tokenized and lemmatized form.

To exemplify, see the graph in Figure 6.2, where the vertices are displayed as domain type and name information with the rest of the attributes being hidden. The AttributedNode features to be extracted in our approach are the v_0 to v_6. While this example is partly a repetition of the one in Chapter 3, we outline it here again for readability purposes.

- $v_0 = \{name : BIBTEX,\ type : EPackage\}$,
- $v_1 = \{name : LocatedElement,\ type : EClass,\ abstract : true\}$,
- $v_2 = \{name : location,\ type : EAttribute,\ eType : EString,\ lowerBound : 1\}$,
- $v_3 = \{name : Bibtex,\ type : EClass\}$,
- $v_4 = \{name : entries,\ type : EReference,\ eType : Entry,\ unique : true,\ \ldots\}$,
- $v_5 = \{name : Entry,\ type : EClass,\ abstract : true\}$.

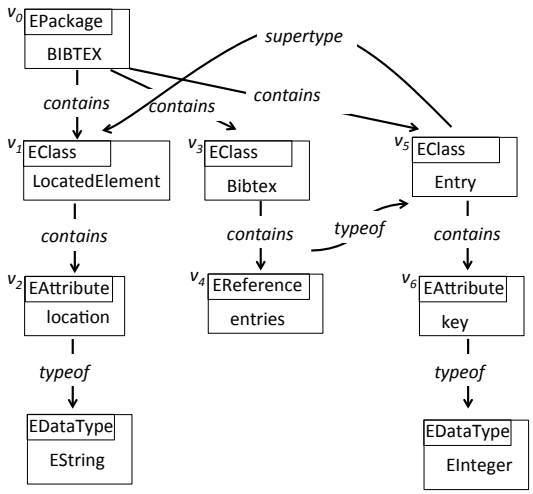


Figure 6.2: A simplified graph representation for (part of) a model.

- v_6 = *{name : key, type : EAttribute, eType:EInteger, lowerBound : 1, upperBound : 1}*.

Note that there are no features corresponding to the basic EDataType leaf nodes in the figure; these are folded into their parent as discussed above. We have seen how we can extract information from (meta)models. In the next section we use the extracted information to encode the structure of metamodels in order to perform clone detection via cluster analysis.

6.4.3 Encoding structure in n-grams and subtrees

Our extended version of SAMOS, supporting subtree extraction in addition to the previously existing options [19] (Chapter 3), has the following three feature settings:

- Ignore the model structure completely, use nodes as is: i.e. the *unigram* setting [22],
- Encode structure in linear chunks: i.e. the *n-gram* (with $n > 1$) setting [19],
- Encode structure in fixed depth subtrees: i.e. 1-depth setting to be described in this chapter.

In terms of the conceptual feature hierarchy in Figure 6.1, one can think of unigrams as corresponding to SimpleFeatures, while n-grams and subtrees (potentially of depth n $>$ 1) are aggregated features containing multiple SimpleFeatures. Based on the graph representation of a model, we can describe n-grams as n consecutively connected vertices. In contrast to [19] which simply omitted the edge information, we also incorporate the edges in the n-grams as SimpleTypes in order to represent the edge information as well. Although it is quite intuitive, the readers are referred to [19] for more details on the graph traversal and n-gram extraction. Some bigrams ($n = 2$) from Figure 2 would be:

- $b_1 = (v_0,\ contains,\ v_1)$,
- $b_2 = (v_1,\ contains,\ v_2)$,

- $b_3 = (v_5,\ hasSupertype,\ v_1)$.

For this chapter, we have extended the structure encoding with another option, namely subtrees. Our motivation was twofold: some shortcomings of n-gram encoding in terms of accuracy (please refer to our first case study in Section 6.5.1 and our previous work [18] for details), and the fact pointed out by Störrle that UML class diagrams are lightly connected trees [167]. We expect this to be the case for metamodels as well, though an empirical assessment remains as future work. Currently SAMOS has been extended for subtree extraction with depth of 1 only (unlike arbitrary n-gram extraction for any n), and hence we show here only 1-depth subtrees. In principle, this can be extended to subtrees of arbitrary depth n $>$ 1 as well, but is left as future work. Similar to the n-grams, based on the underlying graph for a model, we can describe 1-depth subtrees as a root node with all its child vertices. Some 1-depth subtrees from the example are shown in Figure 6.3.

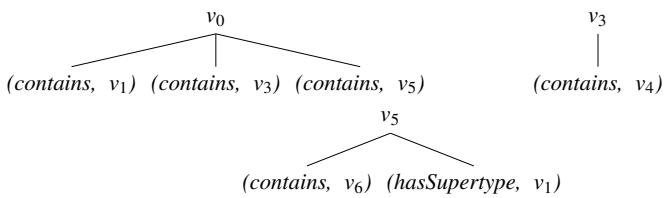


Figure 6.3: Example 1-depth subtrees from Figure 6.2 and vertices above.

The subtrees are extracted in a relatively intuitive form as shown above. Note that the edges are pushed before the vertices (e.g. $(contains, v_1)$), effectively representing each such pair as an n-gram in a single tree node. This is an implementation choice, made due to our wish to use an existing tree edit distance library as a black-box, to be introduced in in Section 6.4.5. A *naive* extraction with edges in separate tree nodes would lead to a default similarity between most subtrees as they naturally have many relations in common—for example, many model elements *contain* other elements. With the containment tree nodes matched by the algorithm, too many subtrees would yield high similarities (for instance, close to 50% similarity merely due to matching edges). We avoid this by collapsing the original tree structure to obtain trees as represented in Figure 6.3. In this way, we can control the n-gram comparison and make sure those with only matching edges do not yield a high similarity value.

6.4.4 Vertex and n-gram comparison

As aggregate features consist of multiple Feature vertices, we first define the extended vertex comparison to account for attributes with the following multiplicative *vertex similarity* formula:

$$vSim(n_1,n_2) = nameSim(n_1,n_2) * typeSim(n_1,n_2) * eTypeSim(n_1,n_2) * attrSim(n_1,n_2) \quad (6.1)$$

$$attrSim(n_1,n_2) = 1 - \frac{\text{\# unmatched attributes between } n_1 \text{ and } n_2}{\text{total \# attr. for that domain type}} \quad (6.2)$$

where *nameSim* is the NLP-based similarity between the names; while *typeSim* and *eTypeSim* are between the domain types and eTypes in the vertices, respectively. The last part *attrSim* measures the similarity for the remaining attributes.

As an example, the number of *unmatched attributes* in Equation 6.2 would equal 1 given a variant of v_4, see Figure 6.2, with *unique* : *false* instead of *unique* : *true*. Note that the simple multiplicative formula in Equation 6.1 might not be optimal, and has issues with certain situations; for example, zero matching attributes leading to zero similarity. We leave it as future work to develop a more suitable scheme. The framework allows relaxing the similarity multipliers: e.g. by instead using a reducing multiplier of 0.5 for non-matching types, ignoring attributes altogether, and so on. N-gram comparison is the semi-relaxed formula [19]; i.e. given n-grams $\vec{v}_1$ and $\vec{v}_2$ with $2n-1$ elements (n vertices and $n-1$ edges corresponding to $v_1^{1..2n-1}$ and $v_2^{1..2n-1}$), the n-gram similarity is:

$$nSim(\vec{v}_1, \vec{v}_2) = ctxMult(\vec{v}_1, \vec{v}_2) * \sum_{i=1}^{2n-1} vSim(v_1^i, v_2^i) \tag{6.3}$$

$$ctxMult(\vec{v}_1, \vec{v}_2) = \frac{1 + |\text{nonzero } vSim \text{ matches between } \vec{v}_1 \text{ and } \vec{v}_2|}{1 + (2n-1)} \tag{6.4}$$

As an implementation workaround, we perform a corrective pass to eliminate *edge-only* matches. For example, SAMOS ignores matches such as A-contains-B vs. C-contains-D when A-C and B-D are completely dissimilar (i.e. $vSim(A,C) = 0$, $vSim(B,D) = 0$); effectively discarding the similarity due to matching merely *contains*. Finally, regarding the NLP part used to compute *nameSim*, SAMOS supports:

- normalizing in cases of mixed-casing and use of other conventions such as camel and snake case,
- tokenization and compound word similarity,
- stemming and lemmatization,
- Levenshtein distance for typos,
- WordNet-based measure of synonymy and semantic relatedness.

The vertex and n-gram comparison, as given above, provide a baseline for comparing more complex structures such as trees. Next we explain two techniques for subtree comparison.

6.4.5 Subtree comparison with ordered tree edit distance

For comparing (meta)model subtrees (of any depth) we have integrated the APTED[7] ordered tree edit distance library [130] into SAMOS. APTED is a state-of-the-art algorithm for computing ordered tree edit distance in a memory efficient manner, using an optimal all-path strategy. As mentioned above, we would like to mostly use the algorithm as a black-box with two minor (plug-and-play) customizations: node data (to contain the n-grams as tree nodes), and the edit distance cost model. The latter is especially important for assigning the cost of minor changes in a non-binary manner; for instance, just changing the abstract modifier of an EClass would lead to a 5% change only. Next we elaborate those two important customization steps for our adoption of APTED.

[7] `http://tree-edit-distance.dbresearch.uni-salzburg.at/`

Data representation We extend the data structure to hold the necessary information in the APTED nodes. Consistently with the subtree examples of Figure 6.3, a node can hold a simple feature (i.e. root) or a bigram (i.e. edge + another simple feature). In both cases each APTED node contains an n-gram.

Cost model APTED uses a cost model to customize different weights (costs) for tree edit operations. We specify the following for the cost of node remove, insert and rename operations used by the algorithm:

- Deleting and insertion with a fixed cost of 1.0,
- Changing (i.e. renaming in the terminology of APTED) a node holding the n-gram p_1 into p_2 with the cost $1 - nSim(p_1, p_2)$. That is, ranging from 0 (exactly same), to 1 (completely distinct). Any value between captures the situation for changing several parts of the node types, attributes and so on.

Pre-sorting the tree nodes As APTED works with ordered trees, we first sort the children of each node with respect to their types, names and attributes. We use a lightweight sorting algorithm at each depth level, hence only consider information at that level. The motivation comes from the fact that in metamodels the order of the child nodes do not matter, and the approach is consistent with the sort-and-compare approach in clone detectors such as NICAD-SIMONE.

Subtree similarity Given two pre-sorted subtrees t_1, t_2 and *size* being the number of nodes in the subtree, we define the edit distance based tree similarity in SAMOS as:

$$tSim_{APTED}(t_1, t_2) = 1 - \frac{APTED(t_1, t_2)}{max(size(t_1), size(t_2))} \tag{6.5}$$

where $APTED$ is the tree edit distance between the two input trees, and *size* yields the number of leaves on a given tree.

Note that we have left the integration of an unordered tree edit distance algorithm (such as Zhang-Shasha [203]) as future work, as a more powerful but costly alternative - especially for subtrees with high depth. Two major reasons would be the lack of an available implementation and the computational complexity. Instead we have developed the (unordered) comparison method for 1-depth subtrees only, as presented in Section 6.4.6. Note that the advantage of using a generic tree edit distance algorithm such as APTED is the uniform applicability for any tree with arbitrary depth.

6.4.6 Subtree comparison with modified Hungarian algorithm

For comparing unordered trees of 1-depth, we employ the Hungarian algorithm [105] for the child nodes. Given 1-depth subtrees t_1, t_2, with parent nodes p_1, p_2 and their respective child node sets C_1 and C_2, we define the Hungarian algorithm based tree similarity in SAMOS as:

$$tSim_{HUNG}(t_1, t_2) = 1 - \frac{vSim(p_1, p_2) + hungarian'(C_1, C_2)}{1 + max(m, n) + (\#\text{ matches in } C_1, C_2 \text{ with } vSim = 1)} \tag{6.6}$$

where we define the *hungarian'* distance, slightly modified over the classical Hungarian distance as:

$$hungarian'(C_1, C_2) = \underset{x}{\operatorname{argmin}} \sum_{c_1 \in C_1} \sum_{c_2 \in C_2} cost'(c_1, c_2) x(c_1, c_2) \tag{6.7}$$

with slightly modified cost function $cost'$ and match function x (as well as additional restrictions) as:

$$cost'(c_1, c_2) = \begin{cases} vSim(c_1, c_2) & \text{if } vSim(c_1, c_2) < 1 \\ 2 & \text{otherwise} \end{cases} \tag{6.8}$$

$$x(c_1, c_2) = \begin{cases} 1 & \text{if } c_1 \text{ and } c_2 \text{ are matched} \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

Given two sets (of possibly unequal sizes), we impose a further restriction to ensure that elements of the smaller set are matched exactly once; while some elements of the larger set can remain unmatched. Furthermore, in the second part of the cost function in Equation 6.8, note the cost of 2 when the two n-grams are completely distinct. We try to mimic the cost of first removing the node and adding a completely new one: represented as two operations instead of a single one (i.e. substitution). We try to find the optimal 1-1 match between all the child nodes of the trees using the vertex similarities for all pairs.

6.4.7 Weighting

SAMOS supports type-based weighting for features. So we can exploit type-based weighting to reduce the relative importance of certain features. Features with more important types (based on our experience) are given higher weights; for instance, matching EClasses should be favored over e.g. matching EParameters. As a larger example, a 1-depth subtree with an EClass node at the root can be allocated a higher weight than a 1-depth subtree with an EOperation root. Note that these weights can be and do get configured according to the model types and problem statement at hand.

For simple types, we use a slight variation of the settings in [22], where we basically give higher weight, for instance, to EClasses, EPackages with lower weight for EReferences, EAttributes and finally even lower for EParameters and EDataTypes. For containment edges in aggregate features, we simply take the average of the type-based weight of all the vertices contained in that feature. For other edges which we consider less significant such as supertypes, we also assign a lower weight. The exact weights are given in Table 6.1.

6.4.8 VSM calculation

SAMOS offers two *modes* for calculating the VSM: linear and quadratic (all-pairs). Linear VSM counts the exact occurrences, that is, computes the total frequencies of features with a single-pass, hence having linear complexity and much faster feature comparison (strict equality exploiting simple hashing). However, this mode is not able to account for synonyms or fine-grained differences in the features such as attributes or types. Quadratic VSM, on the other hand, compares each feature occurrence in the models with the entire feature set of the models, i.e. an all-pairs comparison, with expensive feature comparison such as tree edit distance, leading to a much more expensive (quadratically complex) computation but with proper treatment of fine-grained differences.

metaclass/relation	weight
EPackage	1.0
EDataType	0.2
EClass	1.0
EReference	0.5
EAttribute	0.5
EEnum	1.0
EENumLiteral	1.0
EOperation	0.3
EParameter	0.1
supertype	0.2
throws	0.1

Table 6.1: Table for the type-based weights used in this chapter.

6.4.9 Distance measurement

Originally SAMOS adopted a regular VSM approach: a choice of a distance measure such as cosine and Manhattan, and calculation of the distance over the whole VSM. We have identified several shortcomings in the context of clone detection, and extended the distance measurement. We can summarize our arguments as follows:

- For clone detection, a measure is needed that is normalized (like cosine, and unlike Manhattan distance — as we need relative thresholds), yet size-sensitive (unlike cosine — as size is important). There are several measures in the literature (and implemented in R) that fulfill the requirements of normalization and size-sensitivity, such as Bray-Curtis and Canberra [66].
- Bray-Curtis and Canberra may differ in terms of robustness in certain situations (depending on the dataset, outliers and distribution), but one important distinguishing feature is that Bray-Curtis is weight-sensitive, while Canberra is weight-agnostic. As a result, our weighting scheme, with e.g. higher weights to more important types of model elements, can be realized using Bray-Curtis distance, but not with Canberra.
- VSM assumes orthogonality of the features, and takes all columns into account for distance calculation. This is violated in our case where our features are often (partly) similar to many other features and hence not orthogonal. For clone detection, we limit the distance calculation to the union of *originally contained* features by the two entities, rather than all the features in the dataset. Note that this solves only part of the problem; please see the case studies in Section 6.5 and Secton 6.6 for further discussion.

For the reasons above, we have integrated a masked variant of Bray-Curtis distance (Equation 6.10) into SAMOS, extending the default distance function in the R package *vegan*. Given an N dimensional vector space, data points P and Q for Ecore metamodels representing the whole model, an EPackage or an EClass depending on the scope; P consisting of features $P_1, \ldots, P_m$ and Q consisting of features $Q_1, \ldots, Q_n$, p and q the corresponding vectors on the full vector space for P and Q, the masked Bray-Curtis distance is over the vector subspace $P \cup Q$ (size $\leq m+n$) as:

$$bray'(P,Q) = \frac{\sum_i^{P \cup Q} |p_i - q_i|}{\sum_i^{P \cup Q} (p_i + q_i)}, \tag{6.10}$$

6.4.10 Clustering

As the final step of the SAMOS workflow, we apply clustering on top of the calculated distances to obtain the clone clusters. This in turn can be boiled down to finding non-singleton (size ≥ 2) and sizeable (size $\geq n$) groups of data points that are similar (distance $\leq t$); with n and t thresholds depending on the application scenario. While SAMOS originally supported k-means and hierarchical clustering, we have added and used *Density-Based Spatial Clustering of Applications with Noise* (dbscan) [74]. The algorithm (implemented in R package *dbscan*) uses two parameters, *minimum Points* (i.e. n) and ε distance (i.e. t) to compute density-reachable regions as clusters, with non-reachable regions being labelled as noise. While technical details are beyond the scope of this chapter, dbscan possesses some properties that we desire for clone detection, namely:

- detection of clusters in various (non-convex, non-spherical) shapes,
- detecting noise, i.e. non-clones,
- suitability and efficiency for large datasets.

This concludes the details about extension we have integrated into SAMOS for this work. This extended version of SAMOS will be used in the case studies in the following section.

6.5 Case Studies and Comparative Evaluation

We performed three case studies to evaluate the clone detection capabilities of SAMOS compared to NICAD-Ecore and MACH; in terms of accuracy, and with respect to scalability in the presence of thousands of models:

- **Case Study 1:** We analyzed artificially generated mutation cases and change scenarios where we measured pairwise distances with the base metamodel and the mutated ones to see how sensitive and accurate the different settings of SAMOS are, compared to NICAD-Ecore and MACH.
- **Case Study 2:** At EClass scope, we ran SAMOS with the most accurate setting along with NICAD-Ecore on the configuration management metamodels from ATL Zoo, and comparatively evaluated the clone pairs (and their correct classification, respective precision and recall) found by the two tools.
- **Case Study 3:** We performed a large-scale clone detection exercise, in two-steps with the cheapest and most expensive settings of SAMOS. We aimed to find the metamodel clone clusters in GitHub, for data preprocessing/filtering purposes and future empirical studies on metamodels and DSLs.

For the case studies, we used SAMOS version 0.5, NICAD 3.5 with our Ecore extension 1.1, and finally MACH 0.94. We have put all the relevant material (data, reports, validation/comparison and other information) as supplemental material on our website[7] . We further give the list of settings of SAMOS in Table 6.2.

[7] http://www.win.tue.nl/~obabur/publications/JVLC18/

Component		Setting	Description
Extraction	model	Ecore	extractor for Ecore metamodels
	unit	attributed	key-value pairs for attributes
	structure	uni-/bigram, subtree	different features for comparison
	postprocessing	on	lemma/case standardization
	scoping	model, EClass	whole model, EClasses
Comparison	basic NLP	on	compound-word sim. w/ internal NLP
	advanced NLP	on	WordNet semantic relatedness (Lin)
	type matching	relaxed	0.5 multiplier for non-exact types
	struct. matching	semi-rlx, edit, Hung.	semi-relaxed for bigrams edit dist. & Hungarian for trees
VSM	frequency	sum	sum of frequencies
	idf	off	idf disabled
	weighting	on	type-based weighting (aggregated)
Analysis	distance	msk. Bray-Curtis	masked normalized distance
	clustering	dbscan	density-based clustering
	cut	automatic	automatic clustering with thresholds

Table 6.2: SAMOS configuration for the clone detection case studies.

6.5.1 Case Study 1: Mutation and Scenario Analysis

We have based this case study on a conceptual framework by Stephan et al. for validating our approach [160, 164]. The framework suggests using mutation analysis to evaluate model clone detection techniques. We additionally investigated a few additional scenarios, which can be either conceived as chains of mutations or change scenarios (slightly similar to the ones in Roy [140]). In this section, we detail our assumptions, case design and goals. We finally present the results of SAMOS with different settings, MACH, and NICAD-Ecore; and discuss their potential strengths and weaknesses.

6.5.1.1 Case Design

First we make the simplifying assumption that the scope for this case study is that of EClass clones. Inspecting the Ecore meta-metamodel, we identified a set of mutations, representing noteworthy small changes in an EClass as a starting point. Additionally, we designed some NLP-based cases, a few slightly more complex mutations involving move and swap operations, and finally some more complex scenarios to provide further insight into each tool's shortcomings. We outline our mutation sets as follows.

Set 1a. The first set of cases include 31 metamodels with atomic mutations. The mutations are mostly trivial operations of adding, removing or changing elements.

Set 1b. We added three specific mutations involving subtle element name changes: cosmetic renaming (such as lower vs. upper case, camel vs. snake case) and adding a typo, and replacing a word with its synonym, so that we can evaluate the NLP-related capabilities of the tools.

Set 2. For the second set, we have 5 mutations of reordering, moving and swapping model elements. The regular move mutation involves simply moving a model element elsewhere. In contrast, the extra move mutation (*moveSimilarContainer*) involves moving a model element A contained in B into a distinct container B', with the condition that $vSim(B,B') = 1$ (same vertex similarity)—a change easily detectable using graph-based techniques but possibly not by approximate techniques such as ours. Swap mutations are designed in a similar manner. With

this set, we wish to demonstrate potential shortcomings of SAMOS (in some settings, due to the approximate nature of information representation) and other tools, even if (arguably) some of the cases might not be realistic in real data.

Set 3. The final set of cases includes 4 metamodels obtained from 3 base metamodels by one or more changes: (1) simple renaming of base metamodel elements which lead to a different lexicographic order (as sorting is an essential step of NICAD-* model clone detection and our ordered tree edit distance algorithm), (2) changing the eTypes of all EAttributes in the base metamodel, and (3) removal of one out of many occurrences of model element *A* contained in *E* (*removeNonOrthogonal*), versus (4) removing another model element which has no similarity to the rest of the elements in *E* (*removeOrthogonal*). Some of these may be trivial cases for e.g. detecting of moves using a graph-based comparison, but possibly not for tools such as ours, MACH, and NICAD-Ecore. We aim to investigate cases beyond the very basic mutations.

We used medium-sized base metamodels to be manually mutated as indicated above. To give an impression, the base metamodel for simple mutations contained a single EClass with 5 EAttributes, 2 EReferences and an EOperation to be used for extraction and comparison at the EClass scope. We used the original files for SAMOS and their Emfatic transformations for NICAD-Ecore. For the UML part, we manually replicated all the mutations on UML class diagram representations of the same metamodels in order to evaluate MACH.

6.5.1.2 Goals for the Distance Measures

With respect to accuracy, we set five requirements for distance measures to achieve with SAMOS, and which we advocate as requirements for model clone detection tools in general. We use these to qualitatively evaluate the accuracy of the tools.

- **R1**. Obtain zero distance for mutations leading to Type A clones.

- **R2**. Obtain a non-zero distance for mutations leading to higher level clones such as Type B and C.

- **R3**. For those positive distances in R2, have the distance reasonably small (e.g 0.05 or 0.10 as a breaking point towards Type C) given that all the mutations in this section are small.

- **R4**. For those positive distances in R2, have them matching an intuitive assessment of distance based on the significance or weight of the change, for instance, changing just an EParameter is less significant than changing an EClass.

- **R5**. Overall have *bigger* changes lead to higher distances than *smaller* ones do. For instance, introducing a typo in a name should lead to lower distance than a complete renaming; and changing a type to a lower distance than removing the corresponding element altogether.

In short, we desire accurate but also fine-tuned distances for small changes. This is so because we want to correctly detect cases with accumulated multiple small changes as clones too, such as a complete change of EAttributes types (see Set 3 in Table 6.3), or changing all model element names with snake case into camel case. Overly large distances for small changes, when accumulated across multiple instances, might in fact lead to an incorrect clone classification or missing the clone altogether. This is why we were not satisfied with Set 1 only for evaluation, but also included Set 3. We believe such accumulated changes are interesting and complex scenarios for clones, which have been overlooked by the literature for clone detection evaluation, especially with the mutation-based approaches.

set	id	mutation	unigram	bigram	tr-edit	tr-hung	NICAD	MACH
Set 1a	1	addEClassSupertype	**0.000**	0.042	0.063	0.026	0.100	245 / 510
	2	addEClassEAttribute	0.025	0.049	0.030	0.025	0.100	245 / 510
	3	addEClassEOperation	0.025	0.051	0.145	0.123	0.100	129.5 / 510
	4	addEOperationEException	**0.000**	0.010	0.094	0.061	0.100	**510 / 510**
		...						
	5	changeEClassNameRandom	0.114	**0.737**	0.033	0.027	0.100	**NA**
	6	changeEClassSupertype	**0.000**	0.011	0.055	0.067	0.100	**510 / 510**
	7	changeEClassAbstract	0.026	0.062	0.008	0.007	0.100	476 / 510
	8	changeEAttrNameRandom	0.054	0.005	0.033	0.049	0.100	442 / 510
	9	changeEAttrType	0.040	0.003	0.012	0.010	0.100	**510 / 510**
		...						
	10	removeEClassSupertype	**0.000**	0.048	0.038	0.042	0.100	229.2 / 510
	11	removeEClassEAttribute	0.028	0.054	0.034	0.058	0.100	148.1 / 510
	12	removeEClassEOperation	0.022	0.071	**0.227**	**0.200**	0.100	107.6 / 510
		...						
Set 1b	13	changeEAttrNameCosmetic	0.000	0.000	0.000	0.000	**0.100**	**442 / 510**
	14	changeEAttrNameTypo	0.003	0.001	0.001	0.001	**0.100**	**442 / 510**
	15	changeEAttrNameSynonym	0.001	0.001	**0.068**	0.001	**0.100**	**442 / 510**
		...						
Set 2	16	changeOrder	0.000	0.000	0.000	0.000	0.000	1196/1196
	17	moveSimple	**0.000**	0.001	0.047	0.019	0.060	**1196/1196**
	18	moveSimilarContainer	**0.000**	**0.000**	0.046	0.018	0.060	**1196/1196**
	19	swapSimple	**0.000**	**0.000**	**0.000**	0.007	0.060	**1196/1196**
	20	swapSimilarContainer	**0.000**	**0.000**	**0.000**	**0.000**	0.060	**1196/1196**
Set 3	21	renamingReordering	0.009	0.009	**0.100**	0.008	**0.480**	**130 / 338**
	22	fullRetyping	0.248	0.045	0.101	0.101	**0.530**	**338 / 338**
	23	removeOrthogonal	0.026	0.032	0.030	0.030	0.100	187.2/1066
	24	removeNonOrthogonal	0.051	0.053	0.052	0.052	0.100	220.1/1066

Table 6.3: Pairwise relative distances (reverse similarity) for SAMOS and NICAD-Ecore, absolute similarities for MACH compared to the identity similarity. Values in bold indicate the most problematic cases.

6.5.1.3 Evaluation

We have applied our clone detection technique and report here the distance measure between each case from the three sets and the corresponding base metamodel/UML model. We have used SAMOS with different feature settings (unigrams, bigrams, 1-depth subtrees with ordered tree edit distance, 1-depth subtrees with Hungarian distance) using weighted masked Bray-Curtis distance on the one hand; NICAD-Ecore and MACH on the other hand. Table 6.3 gives a representative subset of the results; for the full set of results, please see the supplemental material[8]. The numbers for SAMOS and NICAD-Ecore represent normalized similarity scores of each case with respect to the base metamodel. Since MACH did not output a normalized distance measure but an absolute one, we reported it along with the identity similarity (base model versus itself) for reference. The most prominent errors are presented in bold in the table.

[8] http://www.win.tue.nl/~obabur/publications/JVLC18/

SAMOS evaluation In general, the results look promising, though not without certain errors and weaknesses for particular settings.

- R1 is not violated by any technique in SAMOS in the two cases with cosmetic changes (mutation 13) and reordering (mutation 16).
- R2 is violated in a number of cases for SAMOS. Unigrams (not unexpectedly, as they ignore structural context) evaluate quite some mutations with zero distance, such as mutations 1, 6, 10 involving supertypes and 4 involving exceptions. Furthermore move/swap mutations (mutations 17-20 in Set 2) are mostly undetected by our technique. Even with the tree settings, SAMOS fails to detect some swap operations: mutations 19 and even 20 for subtree comparison with modified Hungarian algorithm.
- R3 is not uniformly satisfied but the majority of the results are acceptable given a relaxed threshold range of $0.05 - 0.10$. For the mutations involving EOperations (numbers 3 and 12, especially for the tree settings), SAMOS returns relatively large values. While we hope to improve this in the future versions of SAMOS, this is not too problematic considering that removing an EOperation effectively removes all its content as well (EParameters, EExceptions); hence leads to a large distance. Besides, one of the biggest issues, as put in bold in the table, is with bigrams for renaming an EClass (mutation 5): due to the nature of feature extraction (i.e. bigrams), vertices with high number of outgoing edges (such as EClasses typically having many elements) are *over-represented* in the vector space. They are present in many features, hence any change leads to a larger distance in the vector space.
- R4, thanks to the type-based weighting applied, is improved over the non-weighted setting for n-grams as in our previous work [18]: we obtain larger distances for more significant changes, for example, in EClass vs. EAttribute vs. EParameter. For the tree settings however, due to several reasons such as EOperations being represented as separate trees in the VSM, we sometimes end up with non-optimal distances (mutation 12).
- Finally, R5 is relatively well achieved overall. We also would like to highlight the special case with renaming-reordering (mutation 21), which *tr-edit* scores quite dissimilar to the original, due to the lexicographic re-ordering involved. The same problem applies to the case *changeEAttrNameSynonym* (mutation 15), where replacing an EAttribute's name with its synonym leads to a different ordering of the elements in the EClass. Furthermore, element orthogonality (i.e. relatedness of the individual delta elements) affects the pairwise distance in all of the settings (i.e. a fundamental issue with our VSM-based approach), evident in the bottom cases *removeOrthogonal-removeNonOrthogonal* (cases 23, 24).

NICAD-Ecore evaluation NICAD-Ecore overall does a good job in not missing many of the changes. It violates R1 only for the cosmetic EAttribute change case (mutation 13), due to its lack of NLP capabilities. Other than that, it satisfies R1 perfectly with the move/swap cases. Overall it has no problem with R2 too. R3 for the basic cases is satisfied, however the scenarios *renaming-reordering* and *fullRetyping* (cases 21, 22) are not properly assessed by NICAD; respectively due to LCS and line-based granularity for comparison. This line-based (and by default unweighted) approach further leads to the violation of R4 and R5: most changes in this case study are interpreted as "one-line" change, leading to a uniform 0.10 distance; most problematic for the mutations 14 and 15 with very minor changes in element names. Note that although we have not emphasized these cases in the table as a big problem, they lead to quite some inaccuracy (evident in the next case study). Finally NICAD-Ecore is not sensitive to orthogonality at all.

MACH evaluation Inspecting the respective column overall, MACH (with the default setting of the closed-source distributable) seems to be making a few simplifications along the way just like SAMOS with certain of its settings. R1 is only violated for the cosmetic renaming scenario (mutation 13). In the corresponding papers MQ_{lone} is reported to have NLP capabilities, so we suspect they might be turned off in MACH. As for R2, however, MACH seems to be somewhere between the unigram and bigram settings of SAMOS. It seems to ignore or not consistently recognize changes to the following model parts: thrown exceptions (mutation 4), types (of properties, parameters; mutations 9 and others not shown on the table), modifiers, such as the attribute *unique*, and cardinalities. Note that some of these are not shown on the table due to space limitations; please see the supplemental material for details. Since MACH returns an absolute measure of the distances, we choose not to discuss R3-5 (for instance, through a relative measure with respect to the identity comparison, which might lead to a misjudgment). However, we further report that MACH did not find the proper class clone when the class name is changed (*changeEClassNameRandom*, mutation 5) and fails to detect move/swap changes (Set 2). There is also the surprising finding that MACH correctly detected the addition and removal of a supertype, while failing to detect the change thereof (mutation 6). As a final observation MACH, like SAMOS, is sensitive to orthogonality of the changed model elements.

6.5.1.4 Discussion

The mutation and scenario analysis allows us to shed light on the capabilities, simplifications and inaccuracies of the tools compared. Not surprisingly, SAMOS has an increasing accuracy with more complex features and comparison techniques, reaching the best accuracy with 1-depth subtrees and Hungarian distance, at the cost of lower performance and scalability. With NLP as a core capability of SAMOS by design, it handles textual changes better than the other tools. The orthogonality issue is something fundamental to VSM and hence SAMOS with its current design, yet it is surprisingly observed in MACH as well. In any case, this issue is an interesting finding not only in terms of evaluating MACH, but also for the clone detection research in general when evaluating other tools as well. We plan to investigate in the future whether this should be regarded as a positive or negative aspect in clone detection. Finally for SAMOS, we plan to implement specific and more advanced weighting schemes (especially for the tree settings), so that it better meets our goals. As for NICAD-Ecore, it catches all of the non-NLP-related changes, however its line-based granularity and LCS poses a problem. Note that (some of) these might not be fundamental flaws of NICAD itself, but we have taken the standard mode of operation of NICAD-SIMONE and just applied it to metamodels. Yet we believe the problem cannot be properly solved simply by applying pre-/post-processing and normalization steps of the types, modifiers and so on. One might need to change the line-based approach into a more fine-grained comparison.

A final remark is that it is not trivial whether and how NLP could be properly integrated into NICAD (due its use of pre-sorting and LCS), and MACH (due to its use of Prolog indexing and pattern matching). While normalization such as supported by NICAD may avoid pairwise comparison of words with difference casing or typos, it will certainly not be able to avoid pairwise comparisons for more complicated cases such as semantic relatedness of model element names, for instance, using WordNet.

6.5.2 Case Study 2: Metamodels in ATL Zoo

With the results of the first case study giving an idea about the tools, we proceed to the second one where we compare SAMOS with its most accurate setting, *tree-hung* — subtree comparison

with modified Hungarian algorithm, with NICAD-Ecore on a real dataset. We chose NICAD-Ecore over MACH for its higher accuracy, and since MACH lacks several features such as clone clustering, clone classes, and normalized similarity. As the dataset, we chose the conference management metamodels from the ATL Zoo[9], due to the visibility of the public ATL Zoo and the fact that the conference management metamodels form a coherent thematic subset; thus potentially include many clones.

Methodology Given the 14 conference management metamodels from ATL Zoo, we performed the following steps for SAMOS and NICAD-Ecore:

1. Granularity: extract all EClass fragments along with their content (transitive closure of containment),
2. Filtering: remove EClasses without any content or deemed too small (e.g. number of lines < 5 for NICAD-Ecore),
3. Clone detection: On the extracted EClass fragments, run SAMOS with *tree-hung* setting as in case study 6.5.1 and NICAD-Ecore with the distance thresholds: 0, 0.1, 0.3 (respectively Type A, B, C clones),
4. Validation: inspect (manually, but using EMFCompare[10] where possible) random subsets as given below for Type A, B and C. Note down the differences and categorize manually what clone class they belong to. We then report the precision and relative recall (relative to all the inspected validation sets for both tools combined). The validation involved the following sets:
   - Clone pairs common for both, SAMOS $\cap$ NICAD-Ecore,
   - Clone pairs SAMOS $\setminus$ NICAD-Ecore,
   - Clone pairs NICAD-Ecore $\setminus$ SAMOS.

6.5.2.1 Results

Table 6.4 depicts the relevant numbers in our findings. Note that the number of clone pairs for NICAD-Ecore are extracted from the clone clusters (i.e. taking each pair in all the clusters), and differ slightly from the original clone pairs reported by NICAD-Ecore, which are 591 for Type B, and 1054 for Type C (see supplemental material[11] for the clone pair report of NICAD-Ecore). This is due to the fact that NICAD uses connected component analysis for building the clone clusters from pairs. SAMOS also adopts a similar approach with its clustering. A first impression of the table is that SAMOS claimed to detect strictly more clone pairs than NICAD-Ecore for all three clone types, while there was disagreement especially in the B-C categories.

As mentioned, we relied on manual validation to assess the accuracy of the tools. We inspected random subsets of the common and different pairs found by the two tools. For the latter, we focused on the exclusively new pairs only - as by definition a Type A clone pair is a Type B/C clone pair too. We performed a random sampling with confidence level 90% and margin of error 0.10 to get the final sets for manual validation. Table 6.5 gives the resulting set sizes. The validation sets and the manual annotations can be found in the supplemental material. Note that in higher level clones, relatively minor changes are generally not annotated, such as cosmetic changes in Type C clone annotations.

[9] http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore
[10] https://www.eclipse.org/emf/compare/
[11] http://www.win.tue.nl/~obabur/publications/JVLC18/

	Type A		Type B			Type C		
	#c	#p	#c	#p	#ex	#c	#p	#ex
SAMOS	62	702	59	1142	440	40	1687	546
NICAD	58	523	70	599	66	52	1264	665
SAMOS \ NICAD		179		543	405		426	381
NICAD \ SAMOS		0		0	41		0	498
SAMOS ∩ NICAD		523		599	35		1264	167

Table 6.4: Number of clone clusters and pairs found in the conference management metamodels. *c* denoting the number of clusters, *p* pairs and *ex* exclusively new pairs (i.e. Type B \ Type A, and Type C \ Type B).

	Type A	Type B_{ex}	Type C_{ex}
SAMOS \ NICAD	50	58	58
NICAD \ SAMOS	-	26	60
SAMOS ∩ NICAD	60	24	49

Table 6.5: Sizes of the validation sets.

Type A clones SAMOS did a better job in finding the Type A clones, as the dataset includes many cosmetic changes in the model element names and attributes: upper/lower case and extra underscore characters. While the common set for both tools is completely accurate, the difference set is fully in favor of SAMOS. We considered the occasional cases with reordering of elements trivial (and perfectly detected by the two tools), and did not annotate them in our report. We traced the ones missed by NICAD-Ecore in the higher level clone reports and found that NICAD-Ecore detects them with large distances varying from 8 to 73 (i.e. considering each cosmetic change as a one-line difference).

Type B clones The commonly found Type B clones were all correct, containing missing EOpposite values and few additions/removals of elements and supertypes. The exclusively Type B clones found by NICAD-Ecore (i.e. Type B_{ex}(NICAD)), in turn, entirely consists of actually Type A clones misclassified by NICAD-Ecore. Type B_{ex}(SAMOS) has several categories of clone pairs. In the simplest category, there are 51 pairs with very small difference (missing EOpposite, type changed from String to Integer, or interestingly a metaclass change while keeping the element name the same), which are clearly Type B - so correct classification by SAMOS. In other 7 cases there are multiple changes in large EClasses; there, weighting and orthogonality of the changes cause SAMOS to classify those as Type B clone pairs. While in most of these cases the pairs are highly similar to each other, it is difficult to strictly label them as Type B, C or higher. We would like to exemplify a few of those: with a base EClass with elements such as *hasMail, write_paper, submit_paper*, newly added elements such as *hasEmail, submit, write_article* would all be considered highly similar to their base counterparts due to SAMOS' NLP functionality such as typo checking, compound word similarity and WordNet semantic relatedness. Those cause SAMOS to calculate a higher similarity score compared to the addition of orthogonal elements.

While this may be considered controversial—and we have therefore conservatively labeled them as Type C, i.e. misclassified by SAMOS—we believe this to be a powerful feature of our framework.

		Type A	Type B_{ex}	Type C_{ex}
SAMOS	precision	1.00	0.89	0.65
	rel. recall	1.00	0.92	0.78
NICAD-Ecore	precision	1.00	0.46	0.26
	rel. recall	0.75	0.09	0.34

Table 6.6: Precision and relative recall of SAMOS and NICAD-Ecore aggregated from all the validation sets.

Type C clones The changes observed in this category mostly involved multiple addition/removal/changes of model elements, along with above mentioned changes such as cosmetic changes, changes in attribute values (not documented in our annotations). The intersection of the two tools contains mostly correctly classified clone pairs of such type, except 10 cases where additional minor changes, such as a new supertype, led to a high percentage difference in small EClasses.

Type C clones exclusive to NICAD-Ecore are mostly (55 out of 60) misclassified Type A/B clones due to NICAD's line-based granularity and lack of NLP. Two examples are a single supertype addition and a cosmetic renaming, and a type change from String to Integer, both of which lead to NICAD-Ecore classifying those as Type C clones. In a few cases however, NICAD-Ecore does a better job while SAMOS underestimates those as Type B. For the opposite set (SAMOS \ NICAD), all detected pairs are interesting cases with considerable similarity. While most are due to the addition of (multiple) model elements, there were two interesting cases where all the EAttributes of an EClass were replaced with EReferences with the same name and type name (date of eType Date). Nevertheless 16 pairs we manually assessed have relatively too many changes (due to weighting, orthogonality) and hence cannot be considered Type C clones (denoted as >C in our report in the supplemental material[12]). In such cases however, it may be that the elements of the pair are in fact part of a non-spherical (for instance stripe-like) cluster in higher dimensional space, and hence in fact are (indirectly) related. SAMOS performs *reachability clustering* (similar to the connected component analysis in NICAD) and can detect such non-spherical clusters. The implication is, it is not guaranteed that all the pairs in the cluster have low (i.e. lower than threshold) pairwise distances; in fact the ends of the stripe can have much higher distances than the threshold.

Although it is hard to exactly pinpoint the clone type in this set, we annotated the pairs being C, C borderline (arguably C) and >C. We report the accuracy assuming the worst cases for SAMOS, i.e. all borderline cases being incorrect. Table 6.6 gives a final account of both tools' precision and relative recall separately for all the validation sets. The number of real clones (as labelled manually) necessary for relative recall is calculated as the following: (1) find the percentage of the relevant clones in *all* the validation sets, (2) extrapolate the percentages to the original sets with respect to the validation sets (i.e. multiply percentages with the sizes) and (3) average it over all the sets. Note that step (1) means we exhaustively looked at Type B and C sets even when looking for Type A clone pairs, and so on. As for the extrapolation step, we rely on the underlying assumption that random selection of the considerable-sized validation samples ensures similar statistics (percentage of clones in particular) as the population, i.e. the full set of clones, which the sample was taken from.

[12] http://www.win.tue.nl/~obabur/publications/JVLC18/

6.5.2.2 Discussion

In this case study, we performed comparative clone detection on a real dataset. We found evidence to support our claims in case study 1: we need NLP and fine-tuned distances so that multiple small changes accumulate properly to a reasonable sum. NICAD-Ecore overestimates distances due to the lack of these features. On the other hand, SAMOS seems in some cases to underestimate the distance, thus leading to some amount of inaccuracy as well. The results of this case study provided us with insight and helped to improve SAMOS - for instance with better weighting schemes, to begin with. The issue with respect to the orthogonality issue are left as future work to examine in detail, based on interviews with the developers or discussions within the clone detection community. Nevertheless, overall SAMOS performs well with both high accuracy and recall.

Note that we used NICAD with the basic mode of operation as done in SIMONE (i.e. normalization and line-based comparison with LCS). We believe, however, that all of the problems mentioned cannot simply be solved by normalization steps, but rather need more fine-grained (i.e. not line-based) comparison, employing NLP as well. NICAD is nevertheless in principle extensible in the sense that LCS, for instance, can be replaced with other techniques such as set similarity and so on.

We have not reported exact run times here (given the small size of the data). SAMOS is much slower than NICAD: for this case study, the former processes the data in the order of minutes, the latter in just a few seconds. In our case, this level of a performance loss is an acceptable trade-off for the increased accuracy.

For fairness, we would like to emphasize that one of the major weaknesses of SAMOS, namely the move-swap type of changes, are not found in this case study. We leave it as future work to compare the two tools with higher scope and/or model types with typically longer chain-like structure (state charts, or Simulink models already supported by NICAD-SIMONE), which might reveal additional shortcomings of SAMOS. Another interesting line of comparison would involve structural clones (such as blind renamed clones used by SIMONE), which SAMOS does not support at the moment.

6.5.3 Case Study 3 - Metamodel Clones in GitHub

We performed a third case study to apply SAMOS on real data mined from GitHub. The goal here is again to assess the accuracy of our tool, but additionally to demonstrate its scalability to thousands of metamodels. This level of scalability is necessary for empirical studies on metamodels and related DSLs in GitHub (with tens of thousands of items) or industrial DSL/MDE ecosystem evolution studies (with hundreds to thousands of items, and tens of versions per item). Next we detail the mining process and the iterative clone detection we performed.

6.5.3.1 Mining Ecore metamodels from GitHub

For mining GitHub, we did not perform a rigorous methodology as it is not the main focus of this chapter. We rather leave it for the follow-up empirical studies to tackle, along the lines of the related work by Kalliamvakou et al. [92] and Hebig et al. [82]. We do not guarantee a full coverage of *all* the metamodels in GitHub; we are aware that our search mostly covers just the master branches and we are not able to search for files bigger than 384 KB[13].

[13] `https://help.github.com/articles/searching-code/`

Using the web interface of GitHub, we searched for Ecore metamodel files containing the word *EClass*[14]. We crawled all the resulting metamodels and applied two filtering passes, respectively removing (1) files not parsable by EMF or not having any content, and (2) exact duplicates, using MD5 hash comparison.

The search yielded 68,511 code results[15], of which 68,485 could be found and downloaded. Filtering pass 1 removed 1337 files. Roughly 2/3 (45,355) were eliminated after exact duplicate removal, resulting in a final set of 21,793 metamodels. The high number of duplicates contrasts with the results on UML models in GitHub by Hebig et al., who reported merely 12% duplicates. While at first glance one can see that there are public datasets which duplicate the mined metamodels by publishing them again on GitHub, a detailed analysis and explanation of this fact is left for future studies. The clusters of exact duplicates can be found in the supplemental material[16].

6.5.3.2 Iterative Process for Clone Detection: Methodology

The $\sim$22k metamodels still posed a challenge for our framework with computationally expensive settings such as *tree-hung*. While we have put some effort into a distributed (big data) processing back-end for SAMOS [25], it is in a rather early experimental phase. Here we instead aimed to develop an iterative approach and do the assessment. Given various settings of SAMOS, the idea is to start with cheaper, less accurate settings to find clusters with the most promising clone candidates, and process each cluster separately with the more expensive and accurate settings:

1. Run SAMOS with full NLP on unigrams (no structure) of names only (no types, attributes) with threshold 0.3 and the rest of the settings as in the above case studies. However, pre-tokenize and pre-lemmatize all the names to reduce the vector space.

2. Remove the outliers (non-clones), get each clone cluster into a separate bucket.

3. For each bucket, run SAMOS separately with the *tree-hung* setting with attributed 1-depth subtrees (i.e. as in case study 2).

4. Find Type A (i.e. almost the same) and Type C clones (highly similar) with thresholds 0 and 0.3. We choose to omit Type B for simplicity, and the fact that Type C clones already cover a superset of Type B clones.

5. For validating the precision, inspect manually random subsets of the clusters (confidence level 90% and margin of error 0.10) as well as ones with large number of elements, $>$10 for Type A and $>$50 for Type C clones—as cases of higher interest.

6. For validating the relative recall of this new iterative approach (compared to running SAMOS on the whole data at once), run SAMOS with *tree-hung* random subsets assembled from the disjoint buckets and the whole *outlier bucket* to see if we can find clones which would not be found by the iterative approach.

Note that the preprocessed unigram setting in step (1) greatly reduced the vector space, allowing us to handle the whole data set at once. Even with the outlier removal at step (2), we ended up with too many metamodels. We rather processed each bucket separately; hence helping

[14]Using the following URL: `https://github.com/search?q=EClass+extension%3Aecore`, same as what has been done by Kolovos et al. [101]

[15]Performed on June 1st, 2018.

[16]`http://www.win.tue.nl/~obabur/publications/JVLC18/`

tremendously in dealing with the inherently quadratic complexity of our approach. Steps (5) and (6) allowed us to reach qualitative conclusions about the accuracy of SAMOS, and this iterative process. For step (1), we did not go for other approaches such as word stemming (heavier normalization), character n-grams or hash values (such as locality sensitive hashing [182]), in order to be able to use full NLP including WordNet semantic relatedness. An advantage with those approaches, though, is that we could calculate the VSM in a linear way (see Section 6.4.8 for the linear vs. quadratic computation of VSM). Even with the current unigram setting, running SAMOS on the dataset in this case study, but with complex parameters such as NLP turned off and using the linear VSM, only takes minutes (in contrast to hours when using the quadratic VSM).

6.5.3.3 Results

Step 1 of our process resulted in 3813 buckets involving 16048 metamodels, with an average of 4.2 metamodels per bucket. This is yet another important empirical finding: again more than two thirds of the non-duplicate metamodels in GitHub are highly similar to others (provided that SAMOS detects them accurately; see discussion below). We found a few clusters with over 100 items, and one with over 2800. This might be due to the clustering algorithm (cf. discussion in case study 2 on spherical vs. non-spherical clusters), and will be elaborated further in the validation and discussion subsections.

Running *tree-hung* clone detection on the 3813 buckets resulted in 3680 Type A clone clusters containing a total of 8488 metamodels and 2200 exclusively Type C clusters involving 11044 metamodels.

Type A clones Inspecting the 67-item validation set for Type A, we were able to manually label all except one as Type A, hence correctly classified by SAMOS. We identified mostly whitespace differences, reordering of elements, cosmetic changes in names, XML encoding. Changes for instance in nsURI and nsPrefixes of EPackages, generic types and EAnnotations are by design ignored by SAMOS, so we did not consider those as incorrect. A number of cases involved differences in the eType object paths while the names remained the same, where they were treated by EMFCompare as identical as well. One cluster contained metamodels with unresolved proxies (hence eTypes not being retrieved by SAMOS and SAMOS conflating the resulting null values); SAMOS incorrectly labelled it as Type A despite the type differences. A surprisingly large percentage of the cases involved an interesting situation: probably copy-pasted metamodels in different platforms contain different new line characters, resulting in multiple additional *carriage return* characters. This not only happens in a few repositories, but seems like a GitHub-wide phenomenon. This simple fact renders simplistic file comparisons inadequate and might be taken into account by studies like ours (this chapter and [26]) or the ones by Hebig et al. [82] and Kolovos et al. [101] (and replication studies thereof). We additionally checked the 11 clone clusters with more than 10 metamodels per cluster; 9 were correctly classified (except the parts ignored by SAMOS, notably EAnnotations - including any OCL constraints encoded in those): sliced UML metamodels, Petri nets, Ecore meta-metamodel, and various toy example metamodels. In the final two clusters, which we manually labelled as Type B, there were two minor problems: a small implementation error due to empty package not being represented in SAMOS, and token filter removing too short tokens. Note that the latter (together with stopword removal) is in most cases actually beneficial for ignoring insignificant changes; as an example consider the following set of *name_of_class, name_class, name_class_1, name_of_classX* and so on.

Type C clones For validating exclusively Type C clones, we only considered whether the cluster contained *highly similar and interrelated* metamodels (adequate for our follow-up studies); rather omitting the exact categorization into Type B, C or higher. We inspected the 66-item random validation subset plus 13 clusters with sizes larger than 50. The majority of the clusters contained highly similar metamodels, ranging from toy metamodels to ones for C++, Java, fUML, petri nets, business process models. These were situated both intra and inter-repositories. We identified certain flaws of SAMOS as well: apart from a minor implementation error, this involved a case where orthogonality might also be an issue (as discussed in the previous case studies), and two cases where cluster connectivity led to multiple small and consistent sub-clusters getting packed into a large cluster. One of those involved too small metamodels with matching base classes (hence higher weight). In the second, we had a 677-item cluster with UML, SQL, state machine and graph metamodels all thrown in together (presumably because of reachability clustering).

Additional random runs As a result of the 100 runs each with 100 metamodels randomly picked from the buckets and outlier set under the restriction that at most one metamodel per bucket was selected metamodels, we obtained zero Type A and zero Type B clones. However, we found 9 additional clone pairs (i.e. clusters with size two) of Type C. Of these, 4 were correctly classified, while one was borderline and 3 were incorrect. The inaccuracy stemmed presumably from a lot of matching tokens (XElement, YElement, ...) and/or (partly) matching supertypes which were ignored by the unigram pass. In the first case, using WordNet semantic relatedness measures led tokens such as Element and Component to be very similar (Lin similarity of 100%).

6.5.3.4 Discussion and Performance

The results indicate that this iterative mode of SAMOS is able to detect many clones with high precision: 75/78 for Type A, and 76/79 for Type C clone clusters. The high precision is not so surprising given the above case studies used depth 1 subtrees with the Hungarian distance. While we cannot assess the absolute recall, we can claim that the iterative approach also has a very high recall with respect to the non-iterative mode of SAMOS: the 9 small clusters from the random runs are very few compared to the original 4073 clusters with $>$10k models. Yet again a qualitative analysis for identifying the problematic cases (due to implementation error, or something fundamental with SAMOS) helps us to improve SAMOS in the future.

The VSM computation part and the distance calculation being the bottleneck, the complexity of SAMOS is essentially quadratic with respect to the number of features in the dataset. Expensive feature comparison techniques such as Hungarian algorithm and NLP (notably WordNet checks) also greatly increase the execution time. However, as mentioned above, the complexity can be lower when there are many common features across the dataset - reducing the vector space. Thanks to this, we were able to perform the first, unigram pass on a single core of an 2.3 GHz Intel Core i7 processor with 16 GB 1600 MHz DDR3 memory in approximately 14 hours. The second pass, in turn, took close to 12 hours in total for the 3813 buckets. Given the large size of the dataset, we believe SAMOS already achieves a good scalability. We plan to further improve the performance with optimizations and a distributed computing back-end, to cope with tens of thousands of models or even more.

We would like to emphasize that we leave the detailed empirical discussion of the results as future work. We believe it is worthwhile to investigate several phenomena, for instance, whether it is expected to have Type A clones for Ecore meta-metamodels as they are used across many EMF-based projects, and what are the syntactic, semantic and pragmatic differences across the various Petri Net metamodels. We also deliberately left out a comparative performance assess-

ment of SAMOS vs. NICAD-Ecore. This is because we established the higher accuracy of SAMOS in the previous case study, and moved on to assess its scalability and feasibility for a large dataset. In future work, we plan to conduct case studies comparing the performance of SAMOS vs. NICAD-* and MACH as well.

6.6 Overall Discussion and Future Work

In this section we discuss several aspects of our approach and the techniques we developed.

Underlying framework We have built our clone detection technique on top of SAMOS, exploiting its capabilities such as NLP and statistical algorithms for free. The framework easily allowed extension, for example with extraction of new features, and addition of new distance measures. This is one of the strengths of our approach, also considering recent developments within SAMOS such as support for distributed computing and more sophisticated NLP. Using R as the back-end enables us to further experiment with advanced statistical and data mining techniques. We are investigating whether to integrate SAMOS into a data mining framework which would enable us to utilize features such as a GUI, workflow management, and advanced visualization, data mining, and machine learning. We plan to publish SAMOS as an extendible open source framework for model clone detection.

Accuracy SAMOS has a high accuracy, as given in the three case studies. When compared to existing tools, it does a better job, notably thanks to its NLP capabilities (given our clone type categorization). While it is harder to give an account of recall, we tried to give some assessment of (relative) recall in case study 2 (vs. NICAD-Ecore) and 3 (vs. non-iterative execution). The qualitative analysis of the inaccurate cases for all three tools further allow us to pinpoint the weaknesses of the approaches and improve SAMOS where applicable in the future. We believe the variety of case studies and comparison with state-of-the-art clone detectors show SAMOS to be a powerful tool.

Performance and scalability Overall, SAMOS is slower than NICAD-Ecore and MACH, which can mostly be attributed to the quadratic complexity of the all-pairs VSM and distance measurement, the complex NLP employed, and the expensive comparison algorithms performed. Nevertheless, SAMOS can handle quite large datasets (in the order of tens of thousands metamodels, millions of model elements) without too much of a problem, with the help of the iterative process introduced in case study 3, and the distributed computing back-end being developed.

Genericness SAMOS is in principle generic, in the sense that it can be applied to any graph-based model provided one implements the corresponding (metamodel-driven or hard-coded) feature extraction. SAMOS has been or is currently being applied in different contexts for UML class diagrams, industrial DSLs [29], feature models [24] and state charts [198]. It will be further investigated to what extent our technique is applicable to these and other types of models, especially in data flow or block-based languages. Another challenge would be detecting structural clones e.g. for Simulink models, which would certainly require a considerable extension to SAMOS. We regard the detection of semantic clones more of a longer term goal.

Orthogonality and clustering As discussed in all three case studies above, the orthogonality of changes among artifacts changes not only the assessment of (some of the) clone detectors, but

also the assessment of the clone types especially along the borders of Type B, C and higher. It is not clear to us whether orthogonality plays a positive or negative role in the clone classification and perception. We leave it as future work to investigate this phenomenon in detail. Another important aspect is the connectivity clustering of spherical/convex vs. non-spherical/convex shapes. This is a well-known issue in the data mining domain, and we would like to investigate further how it affects the clone detection domain, and whether other clustering techniques might lead to more accurate groupings. For small models, for which even atomic changes lead to too high of a dissimilarity as we found out in our case studies, we plan to investigate solutions such as transforming the similarity function, or using some variant of the *binding strength* advocated by Störrle [168].

Improving SAMOS With the input from the three case studies and our increasing insight working with different types of models, we plan to further extend and test SAMOS with additional features including longer n-grams, arbitrary-depth trees, subgraphs, ordered elements, customized and improved weighting schemes (such as ways to remedy bigram weighting issue, or depth-based weighting), distance measures and statistical algorithms.

More powerful NLP We believe NLP is crucial to tackle real world datasets, and is an underrated aspect in the current model clone detection literature. We plan to further improve the NLP capabilities of SAMOS with part-of-speech tagging, context-based word-sense disambiguation, and more advanced, possibly domain specific semantic resources beyond WordNet.

Other practical aspects Several other practical aspects are reported in the literature as important for applying model clone detection in practice [65, 160]. In this chapter, we have fixed scoping of EClass, EPackage and whole model, thus do not run into the nested clones problem [160]. This would be somewhat important for e.g. EPackage scope, but even more so for other types of models. Other aspects including clone ranking, reporting and inspection, visualization are also left as future work.

Threats to validity A threat to validity of this study is the lack of an assessment of the absolute recall, although we used state-of-the-art (and hence presumably reasonably accurate) clone detector tools as comparison. Ways of further mitigating this problem would be to apply a proper and automated mutation-based assessment tool [162] on the one hand, but also perform further comparative studies with other clone detectors to have a stronger account of the overall relative recall of all the tools combined. The manual labelling of clones, hence the manual validation of accuracy, is also a labor-intensive and error-prone process; to reduce the error rate, we plan to incorporate multiple assessors with techniques from the empirical research domain, and work with domain experts from the industry and the clone detection community for building ground theory [166]. Other additional manual validation techniques could be employed, as proposed by Stephan et al. [163]. Moreover, further comparative case studies not only with NICAD and MACH, but also with additional model clone detectors, notably ConQAT, would lead to a more objective assessment of accuracy. We plan to publish SAMOS along with those validation data in the future for replication studies and reproducibility.

6.7 Related Work

The bulk of model clone detection research follows and is inspired by the larger software clone detection [103] and code clone detection literature [143], which is not covered in this chapter.

The readers are referred to those extensive surveys for further information. Model clone detection research, which can be considered as a sub-area of model comparison [159] in a broader sense, is driven by the major approaches accompanied by tooling as outlined in Section 6.1: CloneDetective/ConQAT [64], SIMONE [9], MACH [167, 168]. Other approaches based on SIMONE are presented by Anthony et al. [16] for UML sequence diagrams and Chen et al. [53] for MATLAB Stateflow models. These approaches use a lot of sophisticated language-specific and problem-specific pre-/postprocessing to extract meaningful model fragments (for instance, conversation patterns in the former) and rely on the mode of operation used in NICAD-SIMONE to find near-miss clones.

Graph-based approaches for MATLAB Simulink models include ConQAT [64] and ModelCD [132]. Both use graph matching and graph-theoretic algorithms to find clones in sets of data flow models. ConQAT is used in comparative studies such as the one by Stephan et al. [162] and is reported to have lower accuracy (especially recall) than NICAD for near-miss clones. ModelCD in turn, contains two algorithms, eScan and aScan, for clone detection, however they are not publicly available. Störrle reports a much higher performance and scalability for MQ_{lone} compared to ConQAT and ModelCD. eScan (for exact clone detection) is implemented by Strüber et al. [171] for detecting model transformation clones and compared against ConQAT. The authors report a much lower performance and scalability than ConQAT for eScan. The performance issue with graph-based approaches is even more evident in more fundamental approaches, namely Similarity Flooding [118] or general (sub-)graph isomorphism.

The aScan approximate clone detection algorithm in ModelCD, however, follows a very similar approach with SAMOS in terms of model fragmentation and vector-based similarity calculation. While we started with IR-inspired scenarios for SAMOS for domain clustering of models, after moving to clone detection we realized that aScan follows a very similar approach to SAMOS. It extracts features for an approximation of the (sub)graphs: n-paths (similar to n-grams in SAMOS) and (p-q)-nodes (similar to depth 1 subtrees in SAMOS). It then uses an occurrence-counting vector with a simple distance measure for calculating the similarity. Nevertheless, we can identify several distinguishing features: they (1) use a very simplistic representation of model information, for instance Simulink nodes by their types only, discarding all other information, (2) do not employ any fine-grained comparison such as NLP, and (3) use additional constraints and heuristics for computing the clones (especially in the case of nested clones). Unfortunately, given the unavailability of ModelCD and the lack of extensive comparative studies of ModelCD with other clone detector tools, it is impossible to assess ModelCD properly. Besides, like MACH, ModelCD is reported on a single model to find the clone within; SAMOS can operate on multiple models and finds clones across different models as well as within the models.

Another approach by Hummel et al. [87] tries to mitigate the shortcomings of fully-fledged graph-based approaches, by indexing subgraphs in terms of their canonical labels approximated by MD5 hashes. Those labels consist of tuples of relevant information such as name of the model files, lists and sequences of normalized statements in the subgraphs. They act as a heuristic step, after which a proper isomorphism check is performed on the potential clone pairs.

Another interesting approach involves clone detection for UML sequence diagrams [110]. The authors suggest flattening sequence diagrams into a 1-dimensional array to construct suffix trees, and identify longest common prefixes of those as clones. Ekanayake et al. present another graph edit distance-based clone detection approach, for business process models [72]. A final interesting reference is the DSL clone detection study by Tairas et al. [175], where the authors investigate Type I and II clones in Object Constraint Language. While the target language is a textual one, it is lifted to the model (abstract syntax) level and clone detection is done via model transformations. The authors note that their approach is not flexible enough to detect Type III (near miss) clones.

6.8 Conclusion

In this chapter we have presented a novel model clone detection approach based on the SAMOS model analytics framework using information retrieval and machine learning techniques. We have extended SAMOS with additional scoping, feature extraction and comparison schemes, customized distance measures and clustering algorithms in the context of metamodel clone detection. We have evaluated our approach using a variety of case studies involving both synthetic and real data; and identified the strengths and weaknesses of our approach along with two other state-of-the-art clone detectors, namely NICAD-SIMONE and MACH. We conclude that SAMOS stands out with its higher accuracy while still being considerably scalable; it proves to be useful for further large-scale clone detection and empirical studies on metamodels and domain specific languages. Regarding model management, detecting and managing (e.g. eliminating) clones is an important process for the health of large scale MDE ecosystems.

Chapter 7

Model Analytics for Industrial MDE Ecosystems

Widespread adoption of model-driven engineering (MDE) in industrial contexts, especially in large companies, leads to an abundance of MDE artifacts such as domain-specific languages and models. ASML is an example of such a company where multi-disciplinary teams work on various ecosystems with many languages and models. Automated analyses of those artifacts, e.g. for detecting duplication and cloning, can potentially aid the maintenance and evolution of those ecoystems. In this work, we explore a variety of model analytics approaches using our framework SAMOS in the industrial context of ASML ecosystems. We have performed case studies involving clone detection on ASML's data and control models within the ASOME ecosystem, cross-DSL conceptual analysis and language-level clone detection on three ecosystems, and finally architectural analysis and reconstruction on the CARM2G ecosystem. We discuss how model analytics can be used to discover insights on MDE ecosystems (e.g. via model clone detection and architectural analysis) and opportunities such as refactoring to improve them.

7.1 Introduction

The increased use of model-driven engineering (MDE) techniques leads to the need to address issues pertaining to the increasing number and variety of MDE artifacts, such as domain specific languages (DSLs) and the corresponding models. This is indeed the case when large industries adopt MDE for multiple domains in their operation. ASML, which is the leading producer of lithography systems, is an example of such a company where multi-disciplinary teams work on various MDE ecosystems involving tens of languages and thousands of models [153]. Automated analyses of those artifacts can potentially aid the maintenance and evolution of those ecoystems. One example issue with these ecosystems is that of duplication and cloning in those artifacts. The presence of clones might negatively affect the maintainability and evolution of software artifacts in general, as widely reported in the literature [94]. In the general sense, when multiple instances of software artifacts (e.g. language or model fragments in our case) exist, a change required in such a fragment (to fix a bug, for instance) would also have to be performed on all other instances of this fragment. Inconsistent changes to such fragments might also lead to incorrect behavior. Therefore, eliminating such redundancy in software artifacts might result in improved

maintainability. While not all cases of encountered clones can be considered negative [94], as some might be inevitable or even intended, it is worthwhile to explore what types of clones exist and what their existence might imply for the system.

The growing number of DSLs in the variety of ecosystems, on the other hand, also demands ways to automatically analyze those languages, e.g. to give an overview of the domains and sub-domains of the enterprise-level ecosystem (i.e. system of ecosystems). Other interesting analyses would include the similarities, conceptual relatedness and clone fragments among the various languages both within and across the ecosystems.

In this chapter, we explore a variety of model analytics approaches using our framework SAMOS [22, 27] in the industrial context of ASML ecosystems. We perform case studies involving clone detection on ASML's data models and control models of the ASOME ecosystem, cross-DSL conceptual analysis and language-level clone detection on three ecosystems (ASOME, CARM2G, wafer handler), and finally architectural analysis and reconstruction, using a technique called topic modelling [44], on the CARM2G ecosystem DSLs. We provide insights into how model analytics can be used to discover factual information on MDE ecosystems (e.g. what types of clones exist, and why) and opportunities such as refactoring to improve the ecosystems.

The rest of the chapter is structured as follows. In Section 7.2 we introduce our main objectives for analyzing MDE ecosystems. In Sections 7.3 and 7.4, we give some background information, respectively on ASML ecosystems and the concept of model clones. We detail how we used and extended SAMOS for the clone detection tasks on ASML's ASOME ecosystem models in Section 7.5. We provide extensive case studies in Section 7.6: clone detection in ASOME data models and control models, cross-DSL conceptual analysis and language-level clone detection, and finally architectural analysis of the CARM2G ecosystem. We continue in Section 7.7 with a general discussion and threats to validity, related work on important topics such as model clone detection and topic modelling in Section 7.8, and finally conclusions and pointers for future work in Section 7.9.

7.2 Objectives

This section presents the objectives that we pursued to analyze the MDE ecosystems at ASML. First, we would like to point out that we used and extended our model analytics framework, SAMOS, to perform various analyses on the MDE artifacts. Since SAMOS already provides a means to detect clones for Ecore metamodels (representing the DSLs in the ecosystems), we explore how this framework can be extended (1) to analyze models adhering to the domain-specific metamodels used at ASML, and (2) to incorporate additional techniques e.g. for architectural analysis.

ASML uses the ASOME modeling language [12] to model the behaviour of its machines. To analyze ASOME models in SAMOS, we first need to understand the elements involved in these models, based on the metamodels they adhere to. This is necessary to extend the feature extraction part, determining e.g. which model parts to extract (and in which specific way) or to ignore. Moreover, while SAMOS defines comparison schemes for the comparison of features extracted from e.g. Ecore metamodels, it has yet to be examined if these comparison schemes are suitable for ASOME models.

Our analysis of ASOME models in this work, namely clone detection, also needs to be evaluated with respect to accuracy and relevance. The goal of clone detection in this context is to find a way to use this information to investigate, and if possible reduce the level of cloning in the models. The largest part of the analysis done in this chapter is clone detection on ASOME

models at ASML. There we considered three aspects: (1) applying and extending SAMOS to detect clones in ASOME models, (2) assessing the accuracy and relevance of the clones found, and (3) improving the maintainability of the MDE ecosystems at ASML based on the discovered cloning information.

Given the variety of MDE ecosystems at ASML, each of which consists of several languages represented by metamodels, we have a few additional objectives related to language-level analyses. First of all, we would like to investigate what overview and high-level insights we can gain by clustering the metamodels of multiple ecosystems. Similarly, we are also interested in the cloning at the language-level within and among the ecosystems, along with their relevance, nature and actionability for improving the quality of the ecosystems. Finally we also consider a focused study on the CARM2G ecosystem only and reconstruct it in terms of the conceptual and architectural layers for architectural understanding and conformance.

The related analyses, addressing the objectives presented above, have been discussed in various sections of the chapter. The extension of SAMOS for clone detection on ASOME models is addressed in Sections 7.5.1 and 7.5.2. The actual clone detection and the interpretation of the results are discussed in the first case studies in Sections 7.6.1 and 7.6.2. The case studies in Sections 7.6.3-7.6.5 on the MDE ecosystems address the rest of the objectives on the language level.

7.3 MDE Ecosystems at ASML

The development of complex systems involves a combination of skills and techniques from various disciplines. The use of models allows one to abstract from the concrete implementation provided by different disciplines to enable the specification, verification and operation of complex systems. However, shortcomings or misunderstandings between the disciplines involved at the model level can become visible at the implementation level. To avoid such shortcomings, it is essential to resolve such conflicts at the model level. To this end, Multi-Disciplinary Systems Engineering (MDSE, used synonymously with MDE in our work for simplicity, although strictly speaking it is a broader domain) ecosystems are employed to maintain the consistency among inter-disciplinary models.

ASML is developing such MDE ecosystems by formalizing the knowledge of several disciplines into one or more domain specific languages [120]. The separation of concerns among the different disciplines helps with handling the complexity of these concerns. Clear and unambiguous communication between the different disciplines is facilitated to enable not only the functioning of the complex system, but also its ability to keep up with the evolving performance requirements. Furthermore, the design flow is optimized, resulting in a faster delivery of software products to the market [12, 120].

In such an ecosystem, concepts and knowledge of the several involved disciplines are formalized into of one or more domain specific languages (DSLs). Each MDE ecosystem has its own well defined application domain. Examples of developed MDE ecosystems at ASML are:

- *ASOME*, from ASML's Software application domain. It enables functional engineers from different disciplines to define data structures and algorithms, and allows software engineers to define supervisory controllers and data repositories [12];
- *CARM2G*, from ASML's Process Control application domain. It enables mechatronic design engineers to define the application in terms of process (motion) controllers (coupled with defacto standard Matlab/Simulink), providing a means for electronic engineers to

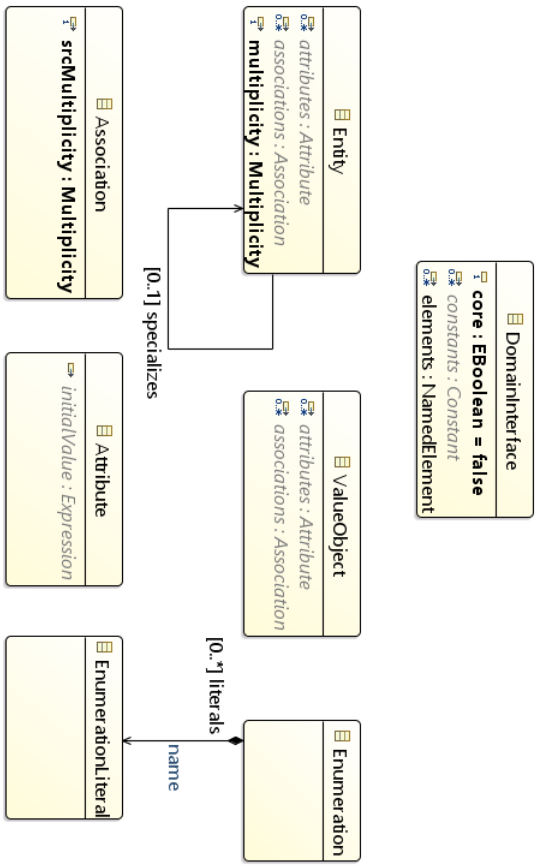


Figure 7.1: Basic elements in a data model

define the platform containing sensors, actuators, the multi-processor, multi-core computation platform and the communication network, and means for software engineers to develop an optimal mapping of the application on to the platform, see [154, 5];

- *Wafer handler (WLSAT)*, from ASML's Manufacturing Logistics application domain. It provides a formal modeling approach for compositional specification of both functionality and timing of manufacturing systems. The performance of the controller can be analyzed and optimized by taking into account the timing characteristics. Since formal semantics are given in terms of a (max, +) state space, various existing performance analysis techniques can be reused [184, 185, 125].

7.3.1 ASOME Models

The ASOME MDE ecosystem is a software development environment that supports the DCA pattern, which separates Data, Control and Algorithms. A motivation to employ this architecture pattern is to avoid changes in the control flow of a system based on a change in data [12]. Using techniques of MDE, ASOME provides metamodels to create data and control models independently of each other.

In the context of DCA and ASOME, data is one of the aspects. Similarly, we also talk about the 'control', 'algorithm' and (overall) 'system' aspects. Within this data aspect, several kinds of systems, interfaces and realizations can be recognized. Domain interfaces and system realizations are just a few examples; other operational examples would include data shifters, services, etc. We further limit our studies on domain interfaces in the data models[1].

Different data elements of an ASML component are represented using one or more data models, the elements of which adhere to a collection of metamodels. As seen in Figure 7.1 data models contain:

- *Domain Interfaces:* any kind of interface in ASOME can express a dependency on another interface. Interfaces allow dependencies among elements originating from other models, hence reuse across models. Within these domain interfaces, several model elements reside including enumerations, entities and value objects.
- *Attributes and Associations:* simple elements with a name, *type*, and multiplicities representing how many instances of these elements can exist at run-time. While attributes are involved strictly in a containment structure, associations can refer to other elements without containing them. Associations additionally distinguish between source and target multiplicities. For collections, the order of the elements contained might be relevant (i.e. list) or not (i.e. set), indicated by the optional flag *order*.
- *Entities:* model elements that contain value objects, attributes and associations to other entities (within the same model or from different models). Entities additionally allow a user to define properties such as deletability, mutability, and persistency.
- *Value Objects:* model elements that contain only attributes of primitive types enumerations or other value objects, but no associations. The concept of value objects has been introduced to be able to avoid repetition.
- *Enumerations:* collections of constant values called *Enumeration Literals*.
- *Primitive Types:* simple types to act as basic building blocks for the ASOME language.

Control models, on the other hand, allow a user to model the flow of control of different components of the system at hand. This is done using state machines. Control models can be of three different types - composite, interface and design[2]. The construction of complex systems in ASOME control models is done using instances of some smaller systems. Composite models contain a decomposition defining what system instances are made up of, along with how they are connected through ports and interfaces. An interface model provides a protocol for a state machine along with a definition of how the system and its interfaces can be defined. A design model uses this protocol to define a concrete realization of this system. Figure 7.2 represents the elements of interest in a control model. These elements are as follows:

- *State Machines:* elements defined in the protocols of interface models or realizations design models. A state machine consists of states, transitions and variables used within these states.

[1] Although, the concept of a data model, in the strict sense, actually does not exist, we simply refer to any such model originating from the data realm as *data model* in this work for simplification.

[2] Composite is strictly a part of the 'system' aspect and not the 'control' aspect. As with data models, the concept of a control model as such does not exist within ASOME. In the scope of this work, we group these types of systems or interfaces and call them control models.

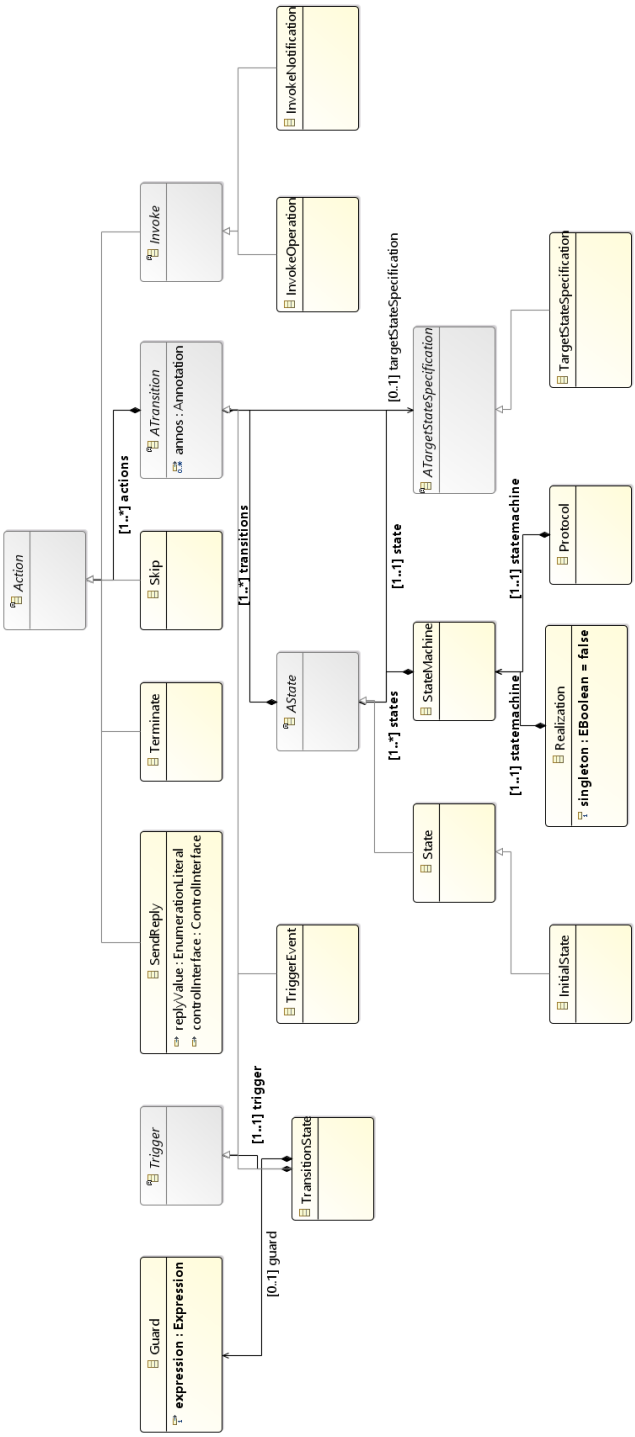


Figure 7.2: Basic elements of a control model

- *States:* elements used to represent different states of the system being modeled using control models. Every state machine consists of a number of states of which one of them is indicated as the initial state.
- *Transition States:* elements that are contained within the state to model the behavior of the system based on different triggers. Each transition state is associated with such a trigger, followed by one or more actions (see below) and optionally a guard (an expression to be solved) and a target state specification[3].
- *Actions:* elements specifying the activity that follows when a triggering event occurs. A sequence of one or more actions is defined in each transition state. These actions could include sending a reply to an interface of a model, terminating the control flow, invoking an operation or a notification, etc.

We will refer to the above basic concepts within ASOME models, when discussing about our approach for clone detection in Section 7.5. While ASOME also facilitates the specification of Algorithm models, these were not considered for the purpose of finding clones in this work. This is due to the fact that there is ongoing effort at ASML to model algorithms and as a consequence there are no models that contain sufficient algorithmic aspects to analyze.

7.4 Model Clones: Concept and Classification

Before detailing the process of clone detection, it is essential to consider what defines a clone. As the core part of the concept and classification is already discussed in Chapter 6, we are skipping it here. However, there are specific concerns for the ASML models, which we address in this section.

For the ASOME data models, the names of elements are considered relevant (argument being that they are similar to conceptual domain models) and the classification of clone types takes changes in the name of model fragments into account. However, for the ASOME control models, since the behavior of these models is analyzed and the structure of the models represents behavior, the classification of clones takes into account the addition or removal of components that modify the structure of the model (in the sense of finding *structural* clones). This is partly in line with the clone category of *renamed clones*, as investigated in the model clone detection literature (e.g. in [53] for Simulink model clones).

7.5 Using and Extending SAMOS for ASOME Models

SAMOS is natively capable of analyzing certain types of models, such as Ecore metamodels. However, it needs to be extended and tailored to the domain-specific ASOME models; this can be considered an extended implementation rather than a conceptual extension. The current section discusses the applicability and extension of SAMOS for clone detection on the ASOME models at ASML. The workflow of SAMOS, as given in Chapter 2, involves the extraction of relevant features from the models. This extraction scheme is metamodel-specific and therefore, an extension to SAMOS is first required, to incorporate a feature extraction scheme based on the ASOME metamodels. SAMOS already uses a customizable workflow for extracting and comparing model elements, e.g. for clone detection. The first step to do this is the metamodel-based

[3]Note that it might be slightly confusing to have a concept mixing the terms *transition* and *state*. In the newer versions of the ASOME language, the name has been changed to *Transition* avoid confusion.

extraction of features, i.e. via a separate extractor for each model type, which is addressed in the following sections.

7.5.1 Feature Extraction

The first step for detecting model clones is to determine the information that is relevant for comparing model elements. In feature extraction, first, the collection of metamodels which jointly define what the Data and Control models adhere to were inspected. Along with input from a domain expert (senior architect and lead language engineer for the ASOME ecosystem) via iterations of face-to-face discussions, we gained insight into the features for each model element that could be considered relevant for clone detection. These include, among others, names and types of the model elements, depending on the particular model element involved. Separate extraction schemes were developed for the Data and Control models.

The above settings describe how a model element (i.e. the vertex in the underlying graph) should be represented as a feature. Next, SAMOS allows a structure setting for feature extraction: unigrams, effectively ignoring the graph structure ; n-grams, capturing structure in linear chunks; and subtrees, capturing structure in fixed-depth trees [27]. These have implications on the comparison method needed (as will be explained in the following sections, and see [27] for details), and on the accuracy of clone detection overall.

The extraction in SAMOS can be specified to treat models as a whole (i.e. map each model to the set of its model elements). In addition, the extraction scope can be narrowed to smaller model fragments, such as extracting features per class in a class diagram. In such cases the analysis done in SAMOS is performed on a model fragment level rather than at the model level, effectively allowing SAMOS to compare and relate model fragments at the chosen scope. For the ASOME models, a number of scopes was investigated. The relevant ones used in the scope of this work are as follows.

Scopes for Data Models Figure 7.1 is a basic representation of the elements contained in the data models. The extraction scopes are listed below:

- *Structured Type and Enumerations:* Similar to the EClass scope we used for metamodel clone detection [27], we treat each entity, value object, and enumeration within a model separately.
- *LevelAA:* This lowest level scope treats each attribute or association separately; hence the name *AA* denoting attributes and associations. These could be considered as one-liner micro clones considered in the code clone detection literature [122].

Scopes for Control Models Figure 7.2 represents the basic elements of ASOME control models. For those models, we considered the following scope:

- *Protocol:* A Control Interface, defined in an interface model, uses a state machine to specify the allowed behavior, i.e. *Protocol*, along with its interface. A Control Realization, defined in a design model, needs to provide a specification that adheres to the Control Interfaces it provides and requires. Similarly, this specification takes the form of a state machine.

7.5.1.1 Domain Specific Concerns for Extraction

A direct (and non-filtered) treatment of the models as their underlying graphs might lead to inaccurate (and noisy) representations, and in turn inaccurate comparison results. We had several domain-specific adaptation for feature extraction of the new model types.

Redundant Information in the Model Graphs Figures 7.3 and 7.4 represent the structure of attributes and associations, respectively, as modeled in the ASOME language. A blind extraction of features along the tree structure for these model fragments would lead to redundant representation of features. For instance, consider a tree-based comparison of any two attributes based on this representation. Since the tree nodes of *Collection* and *Multiplicity* would by definition exist in any attribute, the tree comparison would always detect some minimum similarity (2/7 tree nodes matching). In the extreme case, all attributes with matching multiplicities would have a too high similarity (at least 5/7 tree nodes matching). This would lead to unfair similarities between those model fragments, and is against the *fine-tuned distances* policy of SAMOS [27].

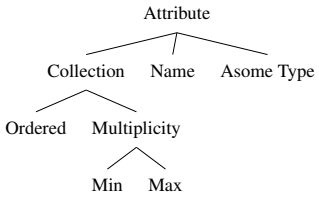


Figure 7.3: Containment structure for *Attribute*.

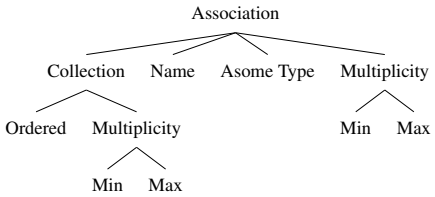


Figure 7.4: Containment structure for *Association*.

To solve this problem, we appended the multiplicity bounds and ordering flag into the attribute or association. Figure 7.5 depicts the new flattened representation for Association. This allows us to have a more meaningful comparison, and in turn more accurate clone detection.

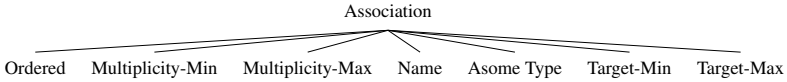


Figure 7.5: Modified containment structure for Association.

Filtering Out Some Model Elements With MDE systems, maintaining traceability between models and eventually derived or generated artifacts, such as code, is important. ASOME uses annotations in Control models to provide this traceability between systems. In Control models, for transition states within a state, such annotations are introduced. During the extraction of features from models, annotations are also extracted. However, the behavior of the model does not depend on these annotations and therefore, including these annotations hamper the accuracy of detecting relevant clones for our interest. To avoid this, the extraction of model features excluded the extraction of these annotations.

7.5.2 Feature Comparison, VSM Calculation and Clustering

While SAMOS has the basic building blocks for the next steps in clone detection, namely feature comparison and VSM construction, we need to specify and extend the comparison needed for

our case studies. The feature comparison setting on the vertex or unit level in SAMOS involve e.g. whether to consider domain type (i.e. metaclass) information of model elements for comparison, and whether and how to compare names using NLP techniques such as tokenization, typo and synonym checking. For this work, we introduced a new option to effectively *ignore* names (i.e. the *No Name* setting). This extension was introduced specifically to find structural clones within ASOME control models, where names do not possess much significance. As for aggregate features containing structural information, such as subtrees (of 1-depth in this work), SAMOS has a built-in unordered comparison technique using the Hungarian algorithm [27]. We employed a specific combination of such settings for various case studies, as will be explained in Section 7.6 per case study.

Building on top of this comparison on the feature level, SAMOS performs an all-pairs comparison to compute a VSM, representing all the models (or model fragments, depending on the extraction scope) in a high-dimensional space. In the case of clone detection, by selecting distance measures (specifically masked Bray-Curtis) and clustering methods (density-based clustering), SAMOS performs the necessary calculations to identify clone pairs and clusters [27].

7.6 Case Studies with ASML MDE Ecosystems

We have performed a wide range of case studies on the models and languages/metamodels used at ASML. In the first two case studies we have detected and investigated the clones in ASOME data and control models, while the others contain language-level analyses on various ecosystems. We are not repeating the explicit tables for SAMOS configurations, but readers can refer to Chapter 5 settings for the conceptual analysis, and Chapters 5 and 6 for clone detection.

7.6.1 Clone Detection in ASOME Data Models

This section discusses the results of the case studies performed using the different settings of SAMOS on the ASOME data models.

7.6.1.1 Dataset and SAMOS Settings

The dataset consists of 28 data models, containing one domain interface each. These domain interfaces in total contain 291 structured type and enumeration model fragments and 574 attributes and associations. Our preliminary runs with the scopes *Model* and *Domain Interface* did not yield significant results, therefore we report here only the lower level scopes. The settings of SAMOS for this case study are as follows:

- *Scopes:* Structured Type and Enumerations, LevelAA.
- *Structure:* Unigrams. For the model fragments at the chosen (low-level) scopes, there is no deep containment structure. So, a unigram representation suffices for this case study.
- *Name Setting:* Name-sensitive comparison, as model element names are important parts of the data models.
- *Type Setting:* A relaxed type comparison (standard setting of SAMOS) for the scope Structured Types and Enumerations and strict type comparison for LevelAA. For the latter, we are interested in strictly similar micro-clones, facilitating easy refactoring.

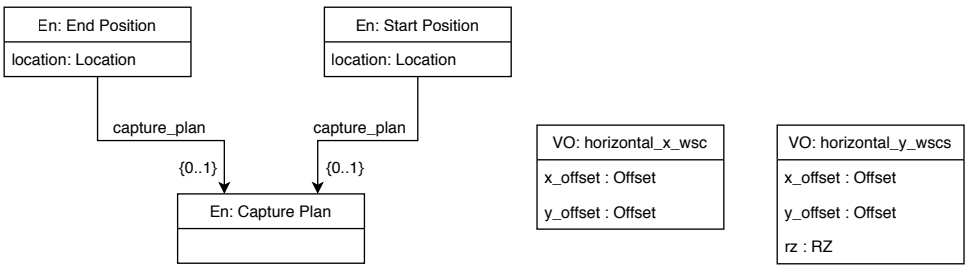
On the given set of data models, using the settings above, we discuss the results we found in the next section.

7.6.1.2 Results and Discussion

This section discusses, per scope, the results obtained through the chosen settings. The discussion is structured as follows: first, the model fragments considered to be clones are discussed; second, the proposition for reducing the level of cloning is presented and finally, the opinion of a domain expert on this proposition is presented.

We found the following clone clusters in the scope of Structured Type and Enumerations:

- *Type A Clones.* Only one clone cluster was found for this category consisting of two Value Objects, named *XYVector*, representing coordinates. This is a small example of duplication in two models, and can be easily eliminated, by reusing from one model in another. The domain expert has commented, that in fact one of the models is actually called *core*, with the intention that it contains the commonalities, while other models import and reuse it.
- *Type B Clones.* We found four clone clusters, each having a single clone pair. We show an example of such Type B clone pairs in Figure 7.6(a), consisting of model fragments with partly similar names (e.g. End Position vs. Start Position) and otherwise the same content. The domain expert's remark was that such cases of redundancy can be considered candidates for elimination via inheritance abstraction. However, due to specific design constraints, and the additional effort to integrate this abstraction in the existing practice, the expert told it is difficult and unlikely that such improvements will be realized.
- *Type C Clones.* We found 23 Type C clone clusters. Figure 7.6(b) shows an example clone pair, with a slight name difference and an additional attribute. Other pairs included changes in names, and attributes. For such clones, redundancy can be eliminated by creating an abstract class with commonalities and extending it. The domain expert commented it is in any case useful to discover such variants in the modelling ecosystem, and it can be investigated which ones can be refactored (along the discussion above for Type B clones).



(a) Type B clone pair example.

(b) Type C clone pair example.

Figure 7.6: Examples for clone pairs in the Structured Types and Enumerations scope for data models.

As for the micro-clones at scope LevelAA, we have the following results:

- *Type A Clones.* We found 53 Type A clone clusters. The most interesting result proved to be the Association *task*, found in a cluster of 9 items. The target of this association is an entity *Task* which belongs to a core data model (a specific instance of a data model which other models depend on). This pattern along with the fact that these associations were all named the same is an indication of consistency and good design, as confirmed by the domain expert. This is an example of the fact that not all clones are harmful and in this case,

the clones are an indication of good design; outliers (if any) can be investigated as an indication of violation of the common practice. As for the duplication in Attributes, the idea of refactoring by lifting these attributes up to a common superclass, was considered by the domain expert with suspicion, due to the additional complexity of introducing inheritance. Duplicate associations in some cases cannot even be eliminated at all, especially in entities from different models.

- *Type B-C Clones.* We found 65 Type B clones clusters, consisting with very similar elements with small differences in e.g. multiplicities. Clones of these types are not candidates for elimination or refactoring however, as remarked by the domain expert. Similarly, among the 81 Type C clone clusters with a higher percentage of differences, the domain expert could not find any good candidate for refactoring. This might indicate that only exact duplicate micro-clones should be considered as useful and actionable, and therefore studied.

Overall Discussion We have provided separate discussions above for our results in different scopes and clone types. A general remark is to be made about the NLP component of SAMOS. In the current setting, due to the tokenization and stop-word removal, SAMOS considers model elements with names *element_m_1* and *element_m_2* as identical; numbers and short tokens are omitted. Moreover, the lemmatization and stemming steps lead SAMOS SAMOS to consider the following as identical or highly similar names: *changed*, *unchanged*, *changing*. In the future we might consider further fine-tuning (and partly disabling) several NLP components considering the problem at hand, when looking for exact clones at the scope of LevelAA.

7.6.2 Clone Detection in ASOME Control Models

This section discusses the case studies performed on control models as well as the results of these case studies.

7.6.2.1 Dataset and SAMOS Settings

The approach taken to detect clones within control models is different compared to the one for data models. This is due to the importance of structure in these models. However, the tree-based setting in SAMOS is still considerably expensive for large datasets. Oh the other hand, a structure-agnostic unigram-based detection with SAMOS [22] would be too inaccurate. Therefore, we follow an iterative approach (similar to Chapter 6). We first narrow down the number of elements for comparison using a cheaper unigram-based analysis. On each cluster found in this first step, we perform a more accurate clone detection separately, therefore reducing the total complexity of the problem. In our previous work [27], we showed that this iterative process leads to only minor drops in recall, but we leave it future work to assess its accuracy in this work.

The data set of control models for this case study contained 691 models, 531 protocols and realizations. A pre-processing step excluded 10 protocols and realizations because these protocols and realizations were very large compared to the other models. Excluding these for the comparison was justified considering it was less likely to find similar models to these ones based on their size. Moreover, these models would slow down the comparison significantly while constructing the VSM. The following settings were chosen for the comparison of control models.

- *Scope:* Protocol level. On this level of comparison, one can compare models based on their behavior, as defined using the state machines residing in these protocols or realizations.

- *Structure:* For the first round of comparisons, the unigram setting was used to find clusters of similar model elements. Fifty such clusters of models were found. The second round of comparison involved inspecting some of the clusters found in the first round. For this round, one-depth subtrees were extracted and compared.
- *Name Setting:* A no-name setting was used for the two rounds of comparison of control models. This was done so we could find models that were structurally equivalent ignoring names.
- *Type Setting:* A strict type setting was used for both rounds of comparisons for control models. In a no-name comparison, to find structurally similar models, this setting allows one to detect model elements with similar types and other attributes.

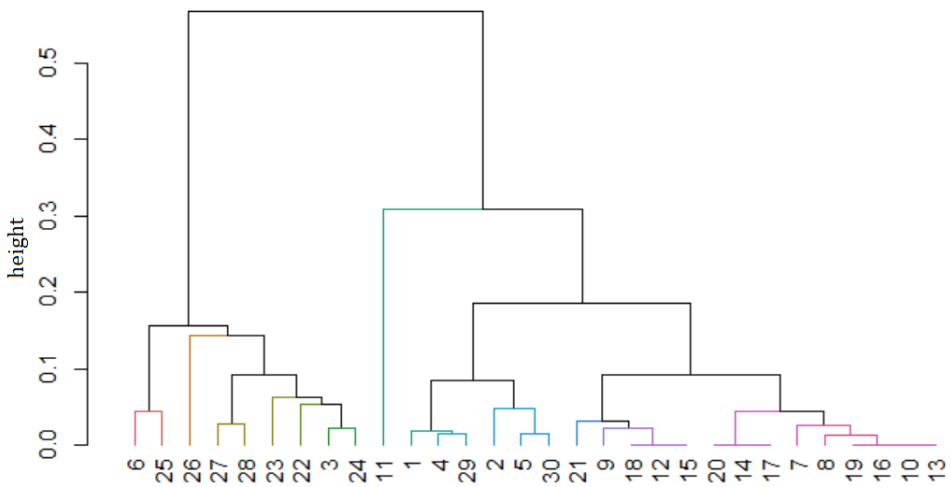


Figure 7.7: A dendrogram representation of cluster 1 with potential clones.

7.6.2.2 Results and Discussion

In this section we provide a detailed discussion and qualitative evaluation on some exemplary control model clone clusters found by SAMOS. Based on the first round, which results in a number of *buckets* with potential candidates for clones, we ran SAMOS with the more accurate subtree setting for a second round of clustering. Figure 7.7 represents the hierarchical clustering of elements contained in one bucket. Note that this hierarchical clustering of elements is used for clarification and discussion purposes only; SAMOS employs a threshold-based automatic cluster extraction technique. The dendrogram represents the Protocol-scope model fragments at each leaf represented by a number, and the vertical axis along with the joints in the tree denote the distance, i.e. dissimilarity, of the fragments.

The models inspected in this cluster were quite large. These contained a single state with a variation in the number and type of transition states, representing an *all-accepting* state. A combination of patterns found in the models is shown in Figure 7.9. The state X contains a number of transition states. The patterns of the different types of transition states found in the models are represented by TS1 through TS6.

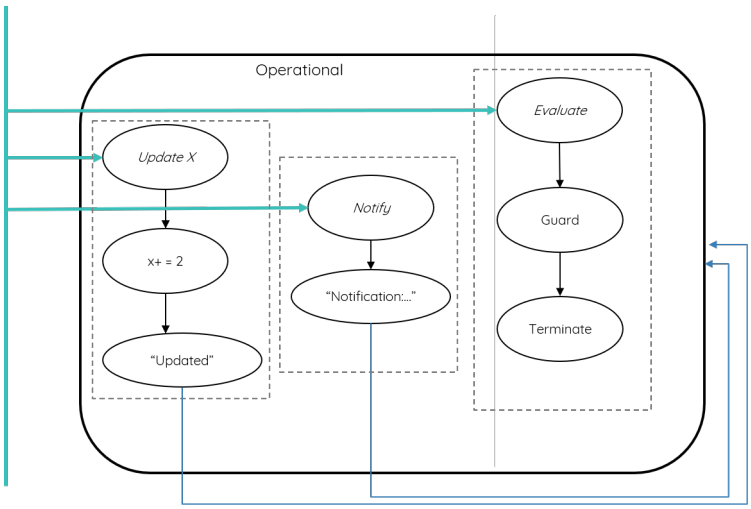


Figure 7.8: Example visualization of some transition states in cluster 1.

Figure 7.8 is an example of a visualization of a few of the transition states in the single state models found in this cluster. The figure shows a single state *Operational* which defines behavior using three transition states. A trigger for each transition state exists. The triggers here are *Update X*, *Notify* and *Evaluate*. Depending on the trigger that has been received, the corresponding transition state is executed. For example, the *Update X* trigger is followed by the action of a State Variable Update where the variable x is updated. Following this, the value "Updated" is sent as a reply. Once the reply is sent, the transition state specifies the same state *Operational* as a target state.

A discussion of the different types of clones, based on the number of occurrences of each type of transition state in the models, is given as follows:

- *Type A Clones.* The elements shown in Figure 7.7 that are represented at the same height and are part of the same hierarchical cluster can be considered type A clones. Examples of such clones are models 20, 14 and 17, and models 18, 12 and 15. An inspection of a collection of models of this type showed that these models had a single state X with multiple transition states. The behavior of the transition states is as shown in Figure 7.9. An inspection of models 19, 16, 10 and 13 showed that they each had 18 occurrences of the pattern TS1; 1 occurrence each of patterns TS2, TS3 and TS4; and 8 occurrences of the pattern TS5. TS6 however, did not occur in these models.

- *Type B Clones.* We can use the dendrogram as guidance to identify elements that are not exactly the same but could be considered similar to each other (up to 10% distance). Model 8, for instance, is similar to the cluster of models 19, 16, 10 and 13; that group is already mentioned above as Type A clones. Model 8 in this case is highly similar to those in the Type A cluster, but contains additionally 2 occurrences of TS5. This makes it a Type B clone compared to rest of the models mentioned.

- *Type C Clones.* Again by inspecting the dendrogram, we can consider, for instance, models 27 and 24 as candidate Type C clones to validate. Figure 7.10 shows the number of times each transition state pattern was found in the models; The number of occurrences are

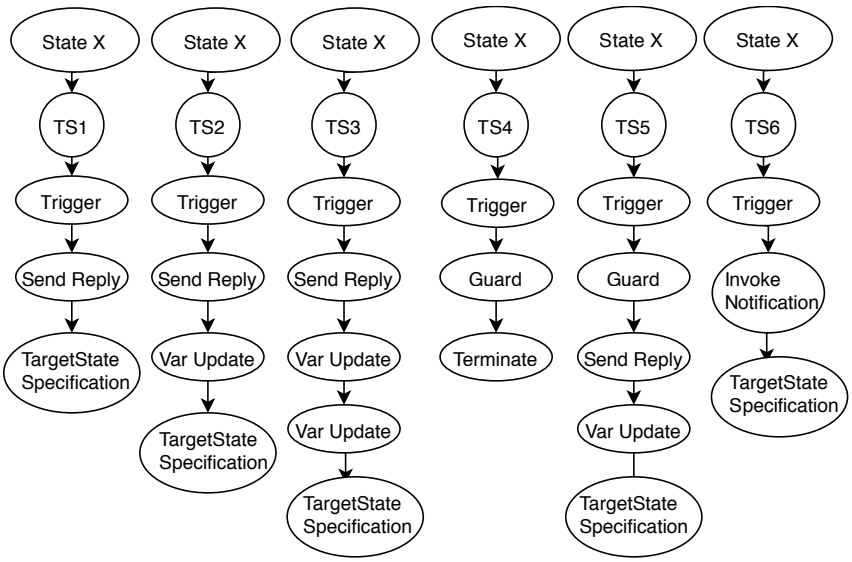


Figure 7.9: Representation of combined behavior of cluster 1 models.

slightly different for four out of six transition state patterns. Therefore, we manually label these as Type C clones as well.

Transition State	Model 24	Model 27
TS1	5	6
TS2	1	1
TS3	1	1
TS4	1	2
TS5	7	4
TS6	2	0

Figure 7.10: Number of occurrences of transition states in models 24 and 27.

We further examined another example cluster to validate the results of SAMOS. Figure 7.11 shows the resulting dendrogram for Cluster 2. The three types of clones in this cluster are discussed as follows. Note that all the models in this cluster share a common pattern (with minor differences as will be discussed below), as shown in Figure 7.12.

- *Type A Clones.* All the elements in this cluster excluding models 3 and 4 can be considered type A clones. The models were all protocols, defining state machines with this structure. The action of sending a reply is associated with a control interface defined in the model. In each of these models, it was observed that the value of the reply sent to the control interfaces was *void*.

- *Type B Clones.* The models excluding model 3 and model 4 could be considered similar to model 4, while not exactly the same. Upon investigating these models, it was noted that the difference between the other models and model 4 is in the action *Send Reply*. While the other models sent an empty reply to the control interface, model 4 replied to the control

interface with a value. Since this is a small percentage of change between these models, model 4 and the models excluding model 3 can be considered type B clones.

- *Type C Clones.* Model 3 can be considered significantly different from the models in this cluster, excluding model 4. The differences between these models is that model 3 was a realization while the other models were protocols. In addition to this, model 3 also sent a value back to the control interface in the *Send Reply* action, like model 4.

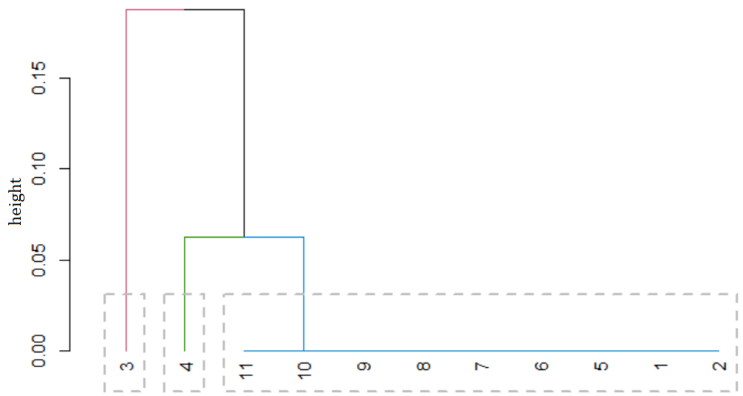


Figure 7.11: A dendrogram representation of cluster 2 with potential clones.

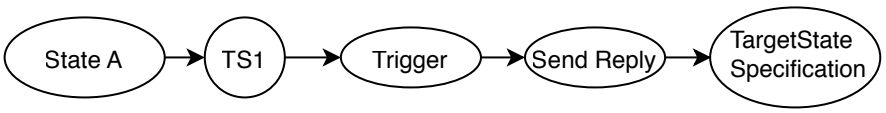


Figure 7.12: Representation of behavior of cluster 2 models.

Overall Discussion The example clusters discussed above represent the types of clusters detected after performing a comparison on the extracted one depth trees representing control models on the 50 unigram-based clusters. Some clusters that were investigated, however, only contained type A clones because all the models found were similar to the other models in that cluster.

While eliminating clones was straightforward for cases in data models, this is not as easy for control models. The presence of duplicates in terms of a sequence of actions might be inevitable if that is the intended behavior of the models. This presents the case for the idea that not all clones can be considered harmful, and some are in fact, intended. However, many occurrences of some transition state patterns have been found in the models. The transition state pattern TS1 as seen in the example cluster 1 (Figure 7.9) was found 18 times each in two inspected models. For such transition states, maybe the language could allow for an easier representation of such a pattern to make it easier for a user to implement this sequence of actions.
According to the domain expert: *"Detecting such patterns of control behavior definitely can be used to investigate whether the user could benefit from a more comfortable syntax. Then an evaluation is needed that needs to take into account:*

1. *Whether the new syntax requires more time to learn by the user.*

2. *Whether the simplification really simplifies a lot (see below).*

For instance, in the example above, even for TS1, the user will need to specify the trigger somehow. In case of a non-void reply, also the reply value will need to be specified. So, TS1 cannot be replaced by one simple keyword. It will always need 2 or three additional inputs from the user. In this case, we will not likely simplify this pattern. However, the way of thinking to inspect whether we can support the user with simplifying the language is interesting. It will always be a trade off between introducing more language concepts vs. writing (slightly) bigger models."

Another suggestion for control models is to investigate the unigram clusters to find the different types of patterns found within the control models. Following this, checking what models do not adhere to these patterns might reveal outliers to investigate, to find unexpected behavior. According to a domain expert:

"I see the line of reasoning and it brings me to the idea of applying machine learning to the collection of models and let the learning algorithm classify the models. Then, investigating the outliers indeed might give some information about models that are erroneous. However, these outliers could also be models describing one single aspect of the system, which would justify the single instance of a pattern. However, I would expect that the erroneous models would also have been identified by other, less costly, means such as verification, validation, review, etc."

7.6.3 Overview on Multiple ASML MDE Ecosystems

As introduced in Section 7.3, ASML has a very diverse conglomerate of MDE ecosystems, developed and maintained by different groups and involving different domains in the company's overall operation. While the architects and managers might have a good idea of (parts of) the enterprise-level big picture, we would like to (semi-automatically) investigate the relation among the different ecosystems with respect to the domains.

Objectives Given the multitude of languages which belong to the various ecosystems, we would like to perform a concept analysis via hierarchical clustering based on the terms used in the metamodels which represent the abstract syntaxes of those languages. Note that we will use the terms metamodel and language interchangeably through our case studies. We have two main sub-objectives in this case study. First we would like to get a good overall picture of the enterprise ecosystem and its compartmentalization into meaningful domains and sub-domains. It is worthwhile to investigate, e.g., whether different ecosystems occupy distinct or intersecting conceptual spaces. Furthermore, it can be interesting to see what close-proximity metamodel pairs or clusters across different ecosystems imply, and whether this information leads to quality improvement opportunities in the ecosystems, such as metamodel refactoring and reuse of language fragments.

Approach To address the objectives above, we process the 86 metamodels belonging to three ecosystems. Using SAMOS, we extract the element names from the metamodels, using the normalization steps including tokenization and lemmatization. We then compute the vector space model over the words, using a tf-idf (with normalized log idf as in [22]) setting also using advanced NLP features such as WordNet checks for semantic relatedness. We then apply hierarchical clustering with average linkage over the cosine distances in the vector space.

Results and Discussion We present our result in the dendrogram depicted in Figure 7.13. Each leaf in the dendrogram corresponds to a metamodel, and all the metamodels are color coded with

respect to their ecosystems. The colored leaves are also projected into the horizontal bar as a complementary visualization. The joints of the leaves and branches can be traced in the y axis, which denotes the distance (dissimilarity) of the (groups of) metamodels. For instance, metamodel pairs in the lower parts of the dendrogram (such as *ds_resource* and *resource*) are very similar. By discussing with the language engineers and domain experts for each ecosystem (three in total, all senior lead engineers for the ecosystems), we gathered a list of remarks that address the objectives above. Next we present a representative summary of those findings, along with key sub-objectives of this case study.

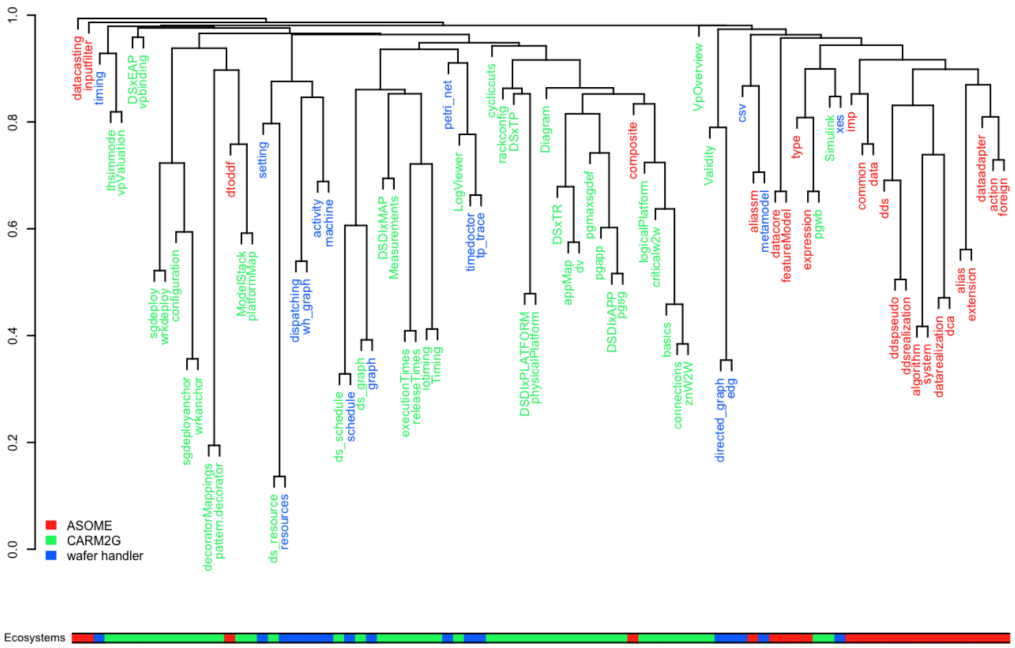


Figure 7.13: Dendrogram depicting the result of clustering the 86 metamodels. Colors denote the ecosystems, while each leaf correspond to a single metamodel in the color-coded ecosystem.

Some remarks involving the general overview, domains and subdomains, proximities across ecosystems would include the following:

- The ecosystems roughly occupy distinct conceptual spaces. As can be followed from the horizontal bar at the bottom, ASOME models are mostly on the right hand side, while wafer handler (less consistently) is in the middle regions.
- There are however small intersections (i.e. impurities in the colored bar, or different colors in the subtrees) among the ecosystems. These are not always surprising or bad because different ecosystems might reuse languages and potentially share sub-domains. However our automated analysis allows having a full overview, in contrast to partial insights of the individual experts.
- Within ecosystems, the domain experts can already detect sub-domains, such as platform, deployment, timing and scheduling for CARM2G, and data for ASOME.

We would conclude the following points regarding highly similar metamodels within and across ecosystems:

- Except two metamodels within CARM2G which are highly similar (height<0.2, i.e. *decoratorMappings* and *pattern.decorator*) — one of which happens to be very small and insignificant, so discarded by the domain experts —, no metamodels within a single ecosystem are too similar. This indicates a healthy design, where each language deals with a distinct conceptual subspace. There are still somewhat similar (height< 0.4) pairs, which might lead to a consideration of a within-ecosystem refactoring. Examples for such pairs would include *sgdeployanchor, wrkanchor* from CARM2G.

- Across the ecosystems, there is a pattern of similar (height<0.4) pairs for CARM2G and wafer handler, specifically for *resources, schedule* and *graph* metamodels. This is apparently due to the fact that wafer handler borrowed these metamodels from CARM2G in early development, while making custom changes as required in time. Our visualization correctly reveals this in a straightforward manner.

- Along the same lines, somewhat similar couples of metamodels across ASOME vs. the other ecosystems exist, though not as significant with the ones above. Examples are *aliassm* vs. *metamodel* which partly contain state machine languages, and *expression* vs. *pswb* which partly contain expression languages.

In summary, according to the feedback we received from the domain experts, such an automated and visual overview of the MDE ecosystems used within a company indeed reveals useful information. This can be used to aid the governance, usage and maintenance of the ecosystems. However, some additional information such as dependencies across languages, the corresponding model instantiations and their relations, usage, etc. could be potentially utilized to further augment our study. Furthermore, we currently cannot detect subtle relations among similar languages which use different terminology. The experts exemplified it by various graph description languages, some of which use the terms *node, edge*; while others use *task, dependency*. This can potentially be mitigated by using a domain-specific thesaurus, in contrast to just relying on general-purpose WordNet for synonyms.

7.6.4 Cross-DSL Clone Detection Across Ecosystems

The concept analysis performed above only deals with the element names, and not the other information in the metamodels such as types, attributes and the structure. It also treats metamodels as a whole. In this case study, we would like to perform a more precise and fine-grained analysis on the metamodel fragments (i.e. sub-parts), in order to reveal similar fragments across, as well as within, the different ecosystems and languages.

Objectives As metamodels across the different ecosystems can have duplicate or highly similar fragments (due to various reasons, e.g. clone-and-own approaches in development or language limitations [27]), we would like to perform clone detection in a more accurate manner, including all the information in the metamodels (not only names). We would like to inspect the clones, their nature (why they occur) and their distribution across the ecosystems. As in the model clone detection case studies, we are also interested to identify potential candidates among these clones which can be used for improving the MDE ecosystems, e.g. in terms of elimination or refactoring.

Approach We considered the 86 metamodels representing three ecosystems in this study. Using SAMOS, we extracted the 1-depth subtrees with full set of model element information from the metamodels, with the EClass scope. Note that we ignored EClasses with no content and supertypes (i.e. zero number of contained elements), assuming they would make less significant

cases for refactoring. We then computed the vector space model over the subtrees, using the *tree-hung* setting [27]. Finally, we applied the clone detection procedure with reachability clustering over the masked Bray-Curtis distances in the vector space.

Results and Discussion Using SAMOS, we found 9 Type A, 13 exclusively Type B (i.e. discarding Type A clusters) and 55 exclusively Type C clone clusters. Table 7.1 gives some of the interesting clusters, which we will discuss next.

id	cluster	t	s	eco
1	dca$LiteralMapping imp$LiteralMapping	A	3	A
2	criticalw2w$BlockName cycliccuts$BlockName	A	2	C
3	ds_resource$ResourceModel resources$ResourceModel	A	1	CW
4	pgwb$PG_LBoundary pgwb$PG_UBoundary	A	1	C
5	physicalPlatform${CoHost,Host}	A	1	C
...	...	...	...	...
6	xes$Attribute{Boolean,Date,Float,...}Type	B	9	W
7	dca$DDTargetIdentifier imp$DDTargetIdentifier	B	5	A
8	ds_schedule$Sequence schedule$Sequence	B	5	CW
9	VpOverview$NXT19{50Ai,60Bi,70Ci,...}Type	B	3	C
10	machine${AxisPositionMapEntry,AxisPositionsMapEntry}	B	3	W
11	{dca,imp,basics}$NamedElement	B	2	AC
12	ds_resource${WorkerResourceSet,IOWorkerResourceSet}	B	2	C
...	...	...	...	...
13	imp$EntityRealizationRecipe imp$EntityRecipe	C	13	A
14	data$Entity datarealization$EntityRealization	C	8.5	A
15	pgsg${HierarchicalBlockGroup,ServoGroupAbstract}	C	6.5	C
16	vpbinding$Binding vpbinding$Clause	C	5.5	C
17	timing$PertDistribution timing$TriangularDistribution	C	4.5	W
18	setting${Location,Motion,Physical,...}SettingsMapEntry	C	4.2	W
19	Validity$ConstrainingNode Validity$ValidatableNode	C	4	C
20	action$IfAction $action$SwitchAction	C	3.5	A
21	{connections,DSDIxPLATFORM,DSxTR,...}$Connection	C	3.4	C
22	expression$UnaryExpression pgwb$PG_UnaryExpression	C	3	AC
23	connections$ConnectionList logicalPltfm$ConnectionBundle	C	3	C
24	pgmaxsgdef$Pgma{BlockAlias,BlockGroup,Block,...}Ref	C	3	C
...	...	...	...	...

Table 7.1: Some of the EClass-scope clones in the metamodels (reported using the convention *metamodelName$EClassName*). **t** denotes the clone type (A, B or C), **s** the average size of the clones in a cluster (with respect to the total number of attributes, operations, etc. for each clone; counting the EClass itself as well) and **eco** the ecosystem the cluster involves: A = ASOME, C = CARM2G, W = wafer handler.

Here is a discussion of Type A clones and opportunities for eliminating duplication:

- There are not too many Type A clones overall and they are quite small (size<3). This indicates little redundancy in general in terms of exact duplication.
- Clusters 1 and 2 show two examples of small clones across different languages, which can be easily refactored and reused.

- Cluster 3 shows an exact clone across ecosystems for the resource language; which we discovered in the previous study to be an evolution/modification from CARM2G to wafer handler.
- Due to our NLP settings (notably ignoring stopwords and typo detection compensating for minor changes), SAMOS finds clone clusters such as 4 and 5 as identical. While they are sigificantly similar and some of these might indicate room for refactoring, the domain experts generally found them to be uninteresting from a maintenance perspective.

As for Type B/C clones and potential refactoring opportunities, we make the following points:

- There is a significant number of Type B/C clones. This indicates that there might be good opportunities to improve the ecosystems.
- Cluster 6 with sizeable (of average size 9) clones shows a clone pattern that we encountered a few more times in this study. According to the domain experts, *xes* is a generated metamodel from xml schemas. There is hence an opportunity to either refactor the xml schema, or the generation process in such a way that the commonality, for instance, is abstracted to a superclass.
- Cluster 7, as well as clusters 13 and 14 show a cloning pattern which happens a few times in the ASOME ecosystem: design vs. realization/implementation. This is a case where clones are *intended*: this pattern is devised to allow the extension of existing language elements for the sake of (a) backwards-compatibility, and (b) clear (conceptual) separation of concerns, i.e. abstract design vs. client-specific implementations.
- Cluster 8 (and some more clusters omitted in the table for conciseness) shows modified fragments of the metamodels adopted from CARM2G into wafer handler.
- Cluster 9 indicates near-duplicate entities for different machine types at ASML. These could be easily refactored, for example, into enumerations, which solves the cloning problem.
- Cluster 10 is an interesting case: the only difference between these metamodels is the multiplicity of an EReference. Domain experts remark that this is an *intended* clone for improving the performance while processing the models in real time in ASML machines.
- Cluster 11 shows small sized clones with a single attribute *name*, differing only in cardinality — considered as a very minor issue, which does not urge for a refactoring. Cluster 12 similarly indicates small clones, which the experts commented they could refactor, for instance, using generics.
- Clusters 15 and 16 show medium-sized clones with common EAttributes and EOperations defined; so the common parts can be abstracted into superclasses. However for the latter, the domain experts remarked that actually there is a superclass which is overridden in subclasses. Due to the limitations of EMF (needing to duplicate the EOperation and pointing to the overridden implementation in EAnnotations), cloning here is supposedly inevitable.
- For the rest of the clusters, the experts indicated a varying degree of usefulness (in terms of refactoring), from low-medium (e.g. in cluster 23 - cardinality difference in small EClasses and in cluster 20 - similar control structures which could be refactored into an abstract superclass) and high (e.g. cluster 17 - one statistical distribution being the ontological superclass of the other smoothed distribution).

Table 7.1 presents the clone occurrences in a flat list. However, we would like to explicitly investigate and visualize the distribution of the clones across languages and ecosystems. To address that, we have constructed the heat map shown in Figure 7.14. It is evident from the figure that there are only a few clones across ecosystems. Notable ones include the resource and schedule languages in CARM2G and wafer handler, parts of expression languages across ASOME and CARM2G and some small basic constructs across all three ecosystems (as discussed above in individual clone clusters). Darker yellow and red parts (i.e. high number of clones) are generally on the diagonal, meaning clones within languages themselves. We can see the reason for these in Table 7.1 e.g. in clusters 9 and 18 with multiple clone pairs. The fact that most clones are within ecosystems is positive, as refactoring across ecosystems might involve multiple developers or teams, projects, and even companies (in the case of outsourcing); hence make it much more difficult and costly to realize.

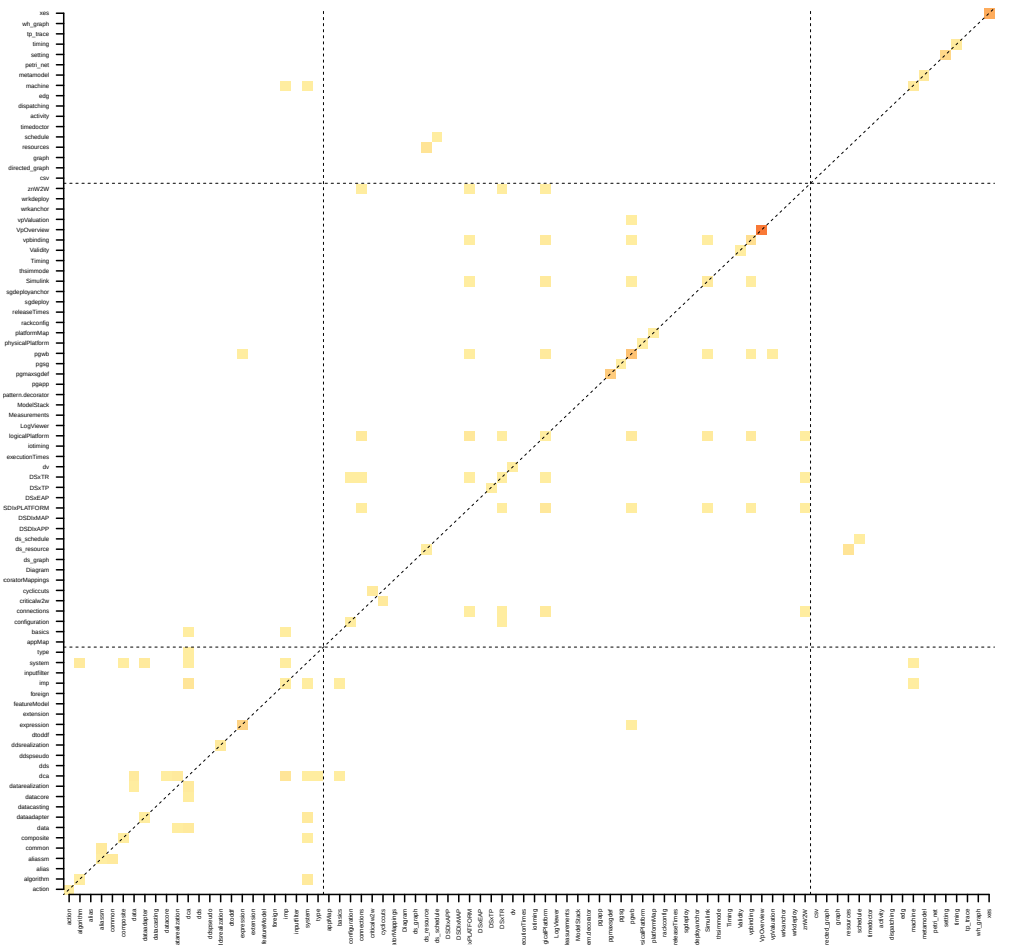


Figure 7.14: Heatmap of the clones across languages and ecosystems. Dashes denote the boundaries of the ecosystems (from left/bottom to right/top: ASOME, CARM2G, wafer handler). Light to dark yellow and red denotes an increasing number of clones.

With the case study in this section, we are able to give both an overview of clones across the ecosystems, and insights into the individual clone clusters and pairs. Overall, the results indicate many opportunities to improve the quality of the enterprise-level MDE ecosystem and its maintenance. Our discussions with the domain experts shed light on specific cases where clones might not only be due to sub-optimal design, but can also be intended (e.g. for performance concerns) or inevitable (e.g. due to language limitations). Our analysis and insights can be used to aid the language design and engineering lifecycle, given the growing number of ecosystems and evolving languages at ASML and other similar companies with large scale MDE practice.

7.6.5 CARM2G Architectural Analysis

The CARM2G ecosystem consists of several architectural layers, as depicted in Figure 7.15. We can regard it as having 5 layers: application, platform, mapping, analysis and deployment, with distinct color coding (given by the domain experts) in the figure. As in the previous case study, the relation between the different layers and sub-languages of CARM2G captured in the 41 metamodels is implicit in the domain expertise of the CARM2G developers. we would like to analyze those metamodels and try to automatically infer useful information with respect to the architecture of the ecosystem.

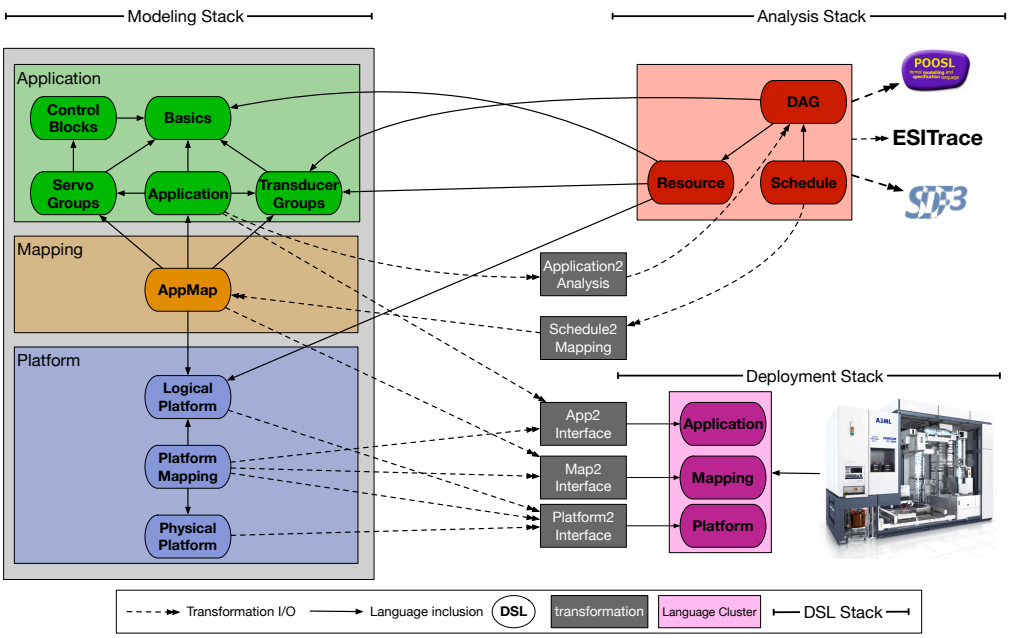


Figure 7.15: CARM2G ecosystem and its architectural layers.

Objectives By topic modelling the terms (i.e. element names) in the metamodels, we aim to reconstruct architectural partitions (arguably layers) and their relation with the individual metamodels. We formulate the following sub-objectives, n being the number of latent topics in the dataset:

1. (unknown n) Identifying how many "topics" there are in the dataset, guessing an optimal n,

2. ($n = 5$) Assessing the correspondence of the automatically mined topics (i.e. partitions) to the architectural layers given by the domain experts,

3. ($n \neq 5$) Identifying additional or redundant partitions or layers by picking different n's.

Approach As in the first case study, using SAMOS we extracted the element names from the CARM2G metamodels, using the normalization steps including tokenization and lemmatization. After removing regular stopwords in English (such as "of", "and") and domain-specific stopwords as determined by the domain experts (such as "name", "type"), We computed a simplistic vector space model over the words in the form of a basic frequency matrix (i.e. no idf). We then performed several experiments with Latent Dirichlet Allocation (LDA, see Section 7.8.3 for details) based on Gibbs sampling [165], to infer the topic-term distributions in the dataset. We did not change the default parameters of LDA (due the exploratory nature of this case study, and the complexity of the parameter setting [129]); we only kept the number of iterations at a relatively high value of 10000 to increase the likelihood of convergence to a global maximum.

Results and Discussion Before going into the results involving topic modelling and the individual sub-objectives, we would like to give a word cloud for the whole ecosystem, as depicted in Figure 7.16. According to the domain experts, this is a very nice summary of CARM2G concepts and can be used, for instance, to describe and document the ecosystem, teach it to new language engineers and modellers.

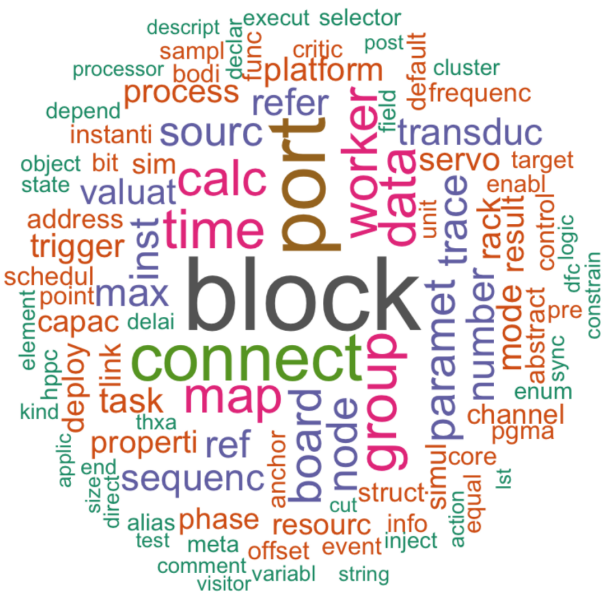


Figure 7.16: Word cloud representation for the whole CARM2G ecosystem.

To address the first sub-objective, we ran LDA with n from 2 to 50, and analyzed the graphs of several metrics in the ldatuning package[4] to investigate near-optimal (minimized or maximized depending on the metric) values for n, as shown in Figure 7.17. We can deduce various near-optimal —while aiming for a small n as much as possible— picks for n: $n = 4, 6$ (Deveaux2014),

[4] https://cran.r-project.org/package=ldatuning

20 (Griffith2004), 6, 10, 16 (CaoJuan2009) and 15, 17 (Arun2010). Two of these metrics have optimum values close to $n = 5$, as given to us by the domain experts, while other metrics predict a larger number of topics. We proceed with $n = 5$, but are going to discuss the implications of picking a lower and higher n later in this section.

After establishing that the number of topics given by the two domain experts is (nearly) agreed on by some of the metrics above, we proceeded with topic modelling with $n = 5$. For the second sub-objective, we are interested in prominent terms per topic, terms by metamodel and the distribution of the topics by metamodel. To evaluate the results, we used a subset of 15 metamodels chosen by the domain experts as key representatives of the CARM2G architectural layers (see Figure 7.15). In Figure 7.18, we present the results of topic modelling specifically for those key metamodels. The interpretation of the figure is as follows. Each row (i.e. y axis) represents a topic (labelled with the top 5 most prominent terms). Each column (i.e. x axis) represents a key metamodel as can also be seen in the legend. The bars at each cell of the matrix represent how likely the metamodels are represented by that topic. Each document is associated with a number of topics, hence the probability values in each column for a specific metamodel add up to 1.

Note that we color coded the metamodels with respect to the architectural layers: green being the application layer, orange mapping, blue platform, purple interface and red analysis. By inspecting the figure along these color codes, for $n = 5$ we can deduce the following:

Topic-1: The first (top-most) topic roughly represents the application layer; mostly represented in four of the application layer metamodels. Nevertheless *basics* and *dv* metamodels, being rather generic and common metamodels, have a mixture of the topics. *pg_wb*, which is originally considered in the CARM2G application layer, does not reside in this layer however, and will be discussed below (referred to as L_{wb}). Note that *DSDIxAPP* from the interface layer mostly covers this topic as well.

Topic-2: The second topic can mostly be associated with the platform layer (most related term:*board*). It is found in the platform layer metamodels (except *platformMap*, to be discussed below) and the *DSDIxPlatform* from the interface layer.

Topic-3: *pg_wb* stands out in the third topic with almost no association with any other topic. This is due to it being a very large and fundamental language describing general purpose language building blocks such as statements and expressions.

Topic-4: The fourth topic does not associate with any of the key metamodels, but potentially (a mixture of) some other niche set of languages in the dataset (e.g. variation point languages). We will discuss this further in our experiments with increasing n, referring to it as outlier layer: L_o.

Topic-5: The final topic has a mixture of mapping and analysis layers. *appMap* naturally is mostly associated with this topic and *platformMap*, *DSDIxMAP* across the other layers as well; all being related. Furthermore, the three metamodels from the analysis layer also consistently reside here.

According to our detailed inspection, and the feedback from the domain experts, we argue that (1) the most prominent terms per topic give only a limited idea about the topics and layers, while (2) the partitioning into topics across languages make a lot of sense. This indeed gives an orthogonal view on the architecture, in terms of the conceptual space. There is still room to change the parameter n for the number of topics, to see whether we can find redundant partitions, and additional (niche) groups of languages besides the standard architectural layers —

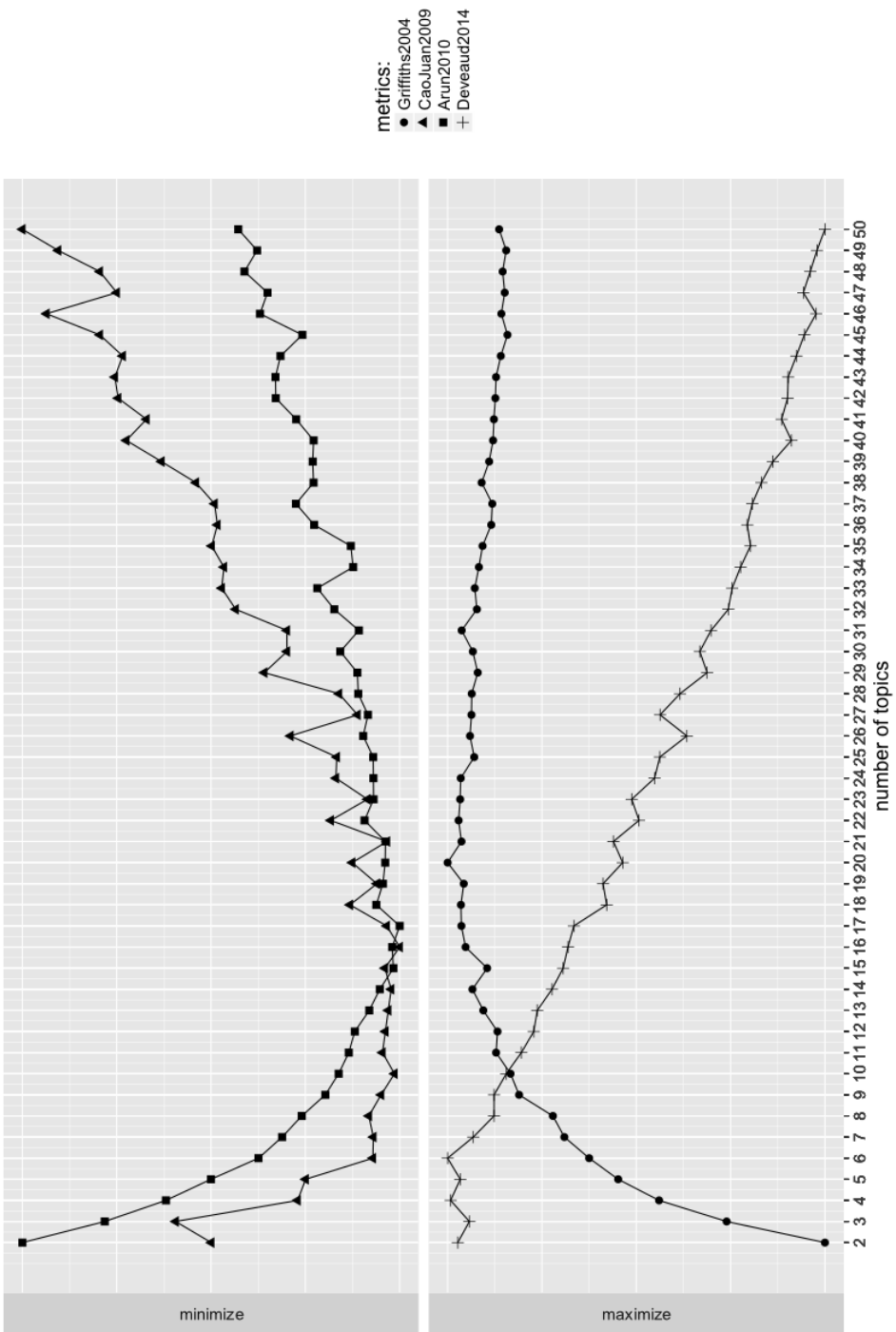


Figure 7.17: Different metric plots for assessing the number of topics.

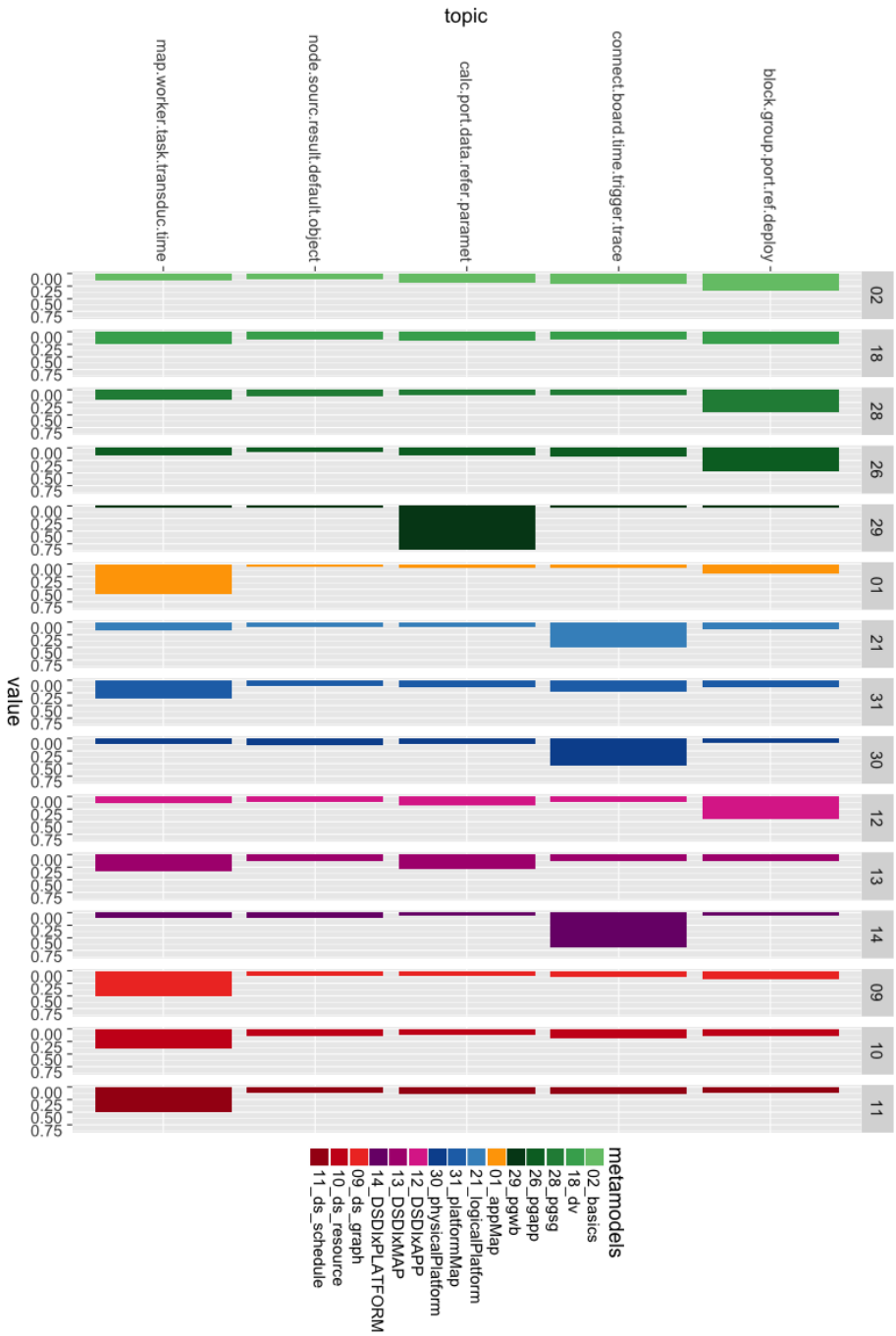


Figure 7.18: Topic distributions per metamodel colored w.r.t. CARM2G architectural layers: green=application, orange=mapping, blue=platform, purple=interface, red=analysis.

addressing the final sub-objective. Following the different near-optimal estimates as discussed above, we remark on the cases with $n = 4, 6, 10, 16$ in text without giving the figures (due to space limitations).

n=4 We obtain a very similar partitioning as for $n = 5$: roughly the application, platform, mapping+analysis layers. The topic which did not correspond to any key metamodel disappears. This might indicate a more optimal partitioning than $n = 5$ if aiming for a high-level layering.

n=6 With larger n, we still get the clear-cut partitions corresponding to platform and mapping+analysis layers. L_o and L_{wb} also remain as is. We see however, that instead of a single application layer, we have two (with divided probabilities for the related metamodels): one with terms *block,group* and other with *connect,port*. These might partly relate to different aspects of an application description.

n=10 We start getting further decompositions: mapping+analysis layer into resource with *platformMap, ds_resource* (terms: *worker, map, resource*); and scheduling with *appMap, ds_graph, ds_schedule* (terms: *task, sequence*, *schedule*). Some of the other partitions, however, start getting a lot fuzzier; platform metamodels for instance are distributed across different topics, *logicalPlatform* is now strongly associated with the application (sub-)topic. However, inspecting all the metamodels involved, we discover further topics, sometimes even represented by a single metamodel: system variants (*vpOverview*), variant binding with a visitor pattern (*vpbinding*), deployment and anchor (*configuration, sgdeploy, wrkdeploy, wrkanchor*) intermixed with simulation (*thsimmode*).

n=16 The topics are further diluted, which makes it very difficult to argue about meaningful partitioning compared to the previous run with $n = 10$.

This exploratory study reveals that we can indeed automatically infer valuable architectural information to a certain extent, as a complementary conceptual viewpoint to architectural layering. It can reveal conceptual partitions in an MDE ecosystem for checking architectural conformance; reveal similar groups and sub-groups of languages; see the cross-cutting concerns across the languages, etc. The accuracy and reliability of topic modelling on the MDE ecosystems, however, is yet to be quantitatively evaluated and further improved. See Sections 7.7 and 7.9 for threats to validity and potential room for improvements in the future.

7.7 Discussion

We have performed a variety of analyses for the MDE ecosystems at ASML. While there have been discussions for each case study separately, in this section we would like to present an overall discussion for our approach.

For the clone detection studies on models, we have extended SAMOS with partly custom-tailored, domain-specific extraction and comparison methods, particular for the ASOME data and control models. The development of these, with the domain experts in the loop, has indicated that the different nature of the (domain-specific) modelling languages, and what the domain experts consider as *relevant* and *irrelevant* pieces of information in the models, are crucial for an accurate, intuitive, and actionable clone detection exercise on those models. These additionally lead to implications on the setting and type of clone detection desired. For example, for the control models, the domain experts were interested in structural clones, while not so much for the data models.

As for the accuracy for the model and metamodel clone detection, we have achieved considerable success in general. However, especially for the structural clone detection for control models, which has been a new extension to SAMOS as introduced in this work, our approach posesses certain shortcomings. We will discuss these as threats to validity later in this section.

For both models and metamodel clones, we have participated in discussions with the domain experts on the nature of the clones, and actionability for improving the MDE ecosystems. Our discussions reveal that some of those clones are indeed harmful and desirable to eliminate or refactor; while others might be inevitable due to language restrictions or even intended, e.g. for certain design goals, performance criteria, or backwards compatibility. Some of those harmful clones are indeed confirmed by the domain experts for potential candidates for improvement, e.g. in the form of refactoring or abstraction. On the other hands, other such harmful clones have been identified as difficult or undesirable to refactor. Reasons for these would include deliberate design decisions (e.g. keeping singleton repositories, as reported in Section 7.6.1) or organizational limitations (e.g. language clones across ecosystems maintained by different teams, as reported in Section 7.6.4).

Interestingly, the results of the clone detection in control models might be used not to refactor the models themselves, but to introduce new language concepts, e.g. in the form of syntactic sugar or abstractions. This could increase the modellers' consistency and efficiency. Nevertheless, there can be certain limitations, such as the additional learning time for the new syntax, and additional modelling effort in the case of abstractions.

Furthermore, we have discovered another use of model clone detection thanks to our discussions with the domain experts. When the cloning pattern is expected and desirable in a certain set of models, we can investigate the occurrence of those clone fragments in all the expected models. *Outlier* models, i.e. expected to have this pattern but not detected in the corresponding clone clusters, might actually indicate inconsistent design. We believe this to be an interesting additional use of SAMOS, and hope to investigate this angle of clone detection in our future work.

Our studies on the system of ecosystems, i.e. the languages and their corresponding metamodels, have been shown to be potentially useful for maintaining the growing and evolving system of ecosystems at ASML. High-level conceptual overview of the enterprise-level ecosystem, as well as finer-grained clone detection on the languages, can provide valuable sources of information in an automatic manner, to understand and monitor the ecosystems, while identifying certain shortcomings of those ecosystems, for instance, in the form of duplication and cloning. The architectural analysis we have performed on the CARM2G ecosystem, on the other hand, can provide a complementary conceptual perspective; in terms of automatic architectural reconstruction and conformance checking with respect to the intended layering. The limitations of the architectural study, a newly explored type of analysis in SAMOS, will be elaborated in the next section as threats to validity.

Threats to Validity Thanks to our extension in this work, SAMOS has been adapted for detecting clones in ASOME data and control models. However, there are several threats to validity for our current implementation. Data models have been compared in a structure-agnostic manner (i.e. using unigrams) at a relatively small scope (i.e. structured types and levelAA; not e.g. the whole model with a deeper containment hierarchy). For larger scopes we would need to use more powerful settings of SAMOS, capturing structure as well (e.g. subtrees, as done for control models).

On the other hand, clone detection for control models has been done on the *Protocol* scope using a similar structure-agnostic setting of unigrams, followed by another comparison using subtrees. The use of one depth subtrees allowed us to reduce the computational time for compar-

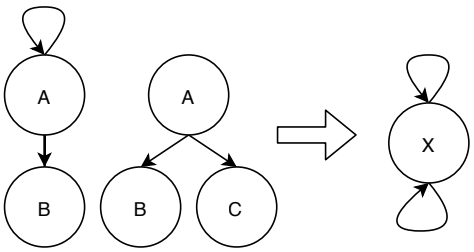


Figure 7.19: Counter example for blind renaming, where SAMOS (erroneously) cannot distinguish between the two cases.

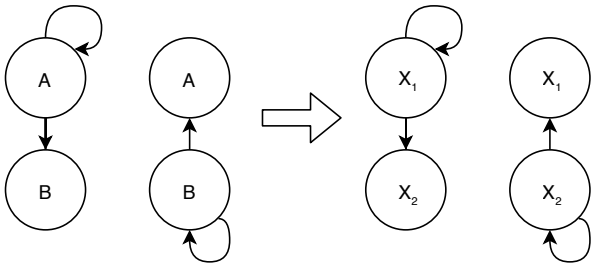


Figure 7.20: Counter example where consistent renaming would be inaccurate.

ison while still including structural information from the models (when compared, for example, to comparing full trees per model fragment). Note that this is still an approximation, and could lead to certain inaccuracies, in which case maybe fully-fledged (and very costly) graph comparison techniques should be employed instead. Obviously there is a trade-off between the accuracy and the running time (hence the scalability) of the selected techniques.

Another issue is with the requirement for selectively employing *ordered* comparison and *unordered* comparison for certain parts of the models. In the current implementation of SAMOS, we have it one way or another for the whole process. ASOME Control models prove to be a mixture of both, where order matters for the list of sequential actions and does not matter for the list of states in a state machine. A selective combination of both would be needed for a more accurate representation and comparison in the case of control models.

The comparison of elements for control models using the *No Name* name setting is similar to the *blind renaming* approach taken in [53]. In such an approach, the identifiers of all the model elements are blindly renamed to the same name, effectively ignoring the relevance of names for the comparison. This approach allows us to find model elements that have similar structure but different values for elements such as guards or triggers or target state specification. While this improves the recall of the results found, the behavior of the two states as shown in Figure 7.19 cannot be distinguished. The two cases on the left side of the figure are treated the same, as depicted on the right side. While the structure is mainly captured by the extracted trees, some structural value is also attached to the names of elements, especially target state specifications. While consistent renaming of model elements might solve this problem, this approach was not taken because the order in which these states are renamed could result in inaccurate comparison results; see Figure 7.20 for an example.

As another important point, we can check what can be borrowed from other research areas (partly or fully) applicable for comparing behavioural models, such as business process mod-

els [69] and finite automata [62]. These could help overcome the inaccuracies our technique currently have with respect to behavioural models.

As for the language analyses presented in this work, several threats to validity exist as well. These would include, for instance, inaccurate NLP for language elements due to the lack of domain-specific dictionaries, cryptic element identifiers, abbreviations and so on. The topic modelling analysis part, however, is treated in a more exploratory manner in this work, in contrast to the domain analysis and clone detection parts which have been validated considerably in previous work. The accuracy and reliability of topic modelling used for the architectural analysis is yet to be studied in detail, quantitatively evaluated and further improved as well. As emphasized in the relevant section, the technique used for topic modelling, namely LDA, is very sensitive to the parameter settings, especially the number of topics. Hierarchical variants of LDA could be investigated to partly overcome this limitation. More specialized topic modelling approaches for shorter bodies of text (e.g. in social media data) could also be experimented with, as the languages in our case also have significantly less content (in the form of metamodel identifiers) than regular text documents.

7.8 Related Work

There are various bodies of literature related to the model analytics case studies in this work. In this section we present those along with brief discussions relevant for this work.

7.8.1 Model Pattern Detection

Model pattern detection is a prominent research area, related to the tasks we are interested in for our research. However, the word *pattern* has been mostly considered synonymous to *design* patterns or *anti-* patterns in the literature [161]. One approach uses pattern detection as a means to comprehend the existing design of a system to further improve this design [180]. This approach involves a representation of the system at hand, as well as of the design pattern to be detected, in terms of graphs. Ultimately, the similarity between the two graphs is computed using a graph similarity algorithm. The paper claims to find (design) patterns within the system even when the pattern has been slightly modified. This approach, however, involves building a collection or catalogue of expected patterns as graphs. Since there were no expectations (by ASML) of the kind of patterns that needed to be detected in our case, we focused on finding e.g. model clones in an unsupervised manner as discussed in the section below.

7.8.2 Model Clone Detection

While code clones have been previously explored in abundance and hence can be associated with some standard definition and classifications [143, 103], relatively less work has been done in the field of model clone detection, resulting in the lack of such clear definitions. Model clones have been defined as *"unexpected overlaps and duplicates in models"* [138]. Störrle discusses the notion of model clones in depth, as *a pair of model fragments with a high degree of similarity between each other* [167]. Model fragments are further defined as model elements closed under the containment relationship (the presence of this relationship between elements implies that the child in the relationship cannot exist independently of its parent).

Quite a few approaches advocate representing and analysing models with respect to their underlying graphs, for clone detection purposes. One such approach involves representing Simulink

models in the form of a labeled model graph [64]. In such graph-based methods, the task of finding clones in the models boils down to finding similar sub-graphs within the constructed model graph. To do this, all maximal *clone pairs* are found within the graph (with a specification as to what constitutes a clone pair in their case). The approach of finding these maximal clone pairs is NP-complete and to reduce the running time, [64] the approach is modified to construct a similarity function for two nodes as a measure of their structural similarity. Finally, the detected clone pairs are aggregated using a clustering algorithm to find the resulting *clone classes* in the model. The disadvantage of this approach however, is that *approximate* clones are not captured.

The work presented in [84] compares block based models by assigning weights to relevant attributes for comparison such as names, functions of the block and interfaces. A similarity measure is defined to assign a value for the comparison and this value is stored for every pair of blocks being compared. This approach is taken to find variability in models in the automotive domain. Variations were introduced to a base model to add or remove functionality. By inspecting the similarity values, one could find models similar to a selected base model. SAMOS also uses the idea of computing similarity using a vector space model to represent the occurrence of features in each model.

Stö provides a contradictory notion however, that for some UML models, the graph structure may not necessarily be the most important aspect of the models to consider for clone detection [167]. Section 4.2 discusses that for some UML models, most of the information worth considering resides in the nodes as opposed to the links between these nodes. Therefore, the approach taken in this paper defines the similarity of model fragments as the similarity of the nodes in such model fragments instead of the similarity of the graph structure of these model fragments. To construct this measure of similarity, the approach involves using heuristics based on the names of the elements being compared. Such an approach is justified when considering that *"most elements that matter to modelers are named"* [167]. This approach works for models where structure does not represent much in terms of model behavior. However, when the behavior of the models is represented in terms of structure, this approach cannot be used.

7.8.3 Topic Modelling

Topic modelling, an approach in Information Retrieval and Machine Learning domains, involves a set of statistical techniques in text mining to automatically discover prominent concepts or topics in natural language text document collections [165, 44]. Topics are typically conceived as collections or distributions of frequently co-occurring words in the corpus, which are assumed to be often semantically related. Topic models are often employed as an effective means to work on unstructured and unlabelled data such as natural language text, to infer some latent structure in the form of topic distributions (over the documents) and term distributions (over the topics).

Topic Modelling Applications for Software Engineering Besides in text mining tasks, topic models are used in other disciplines, such as bioinformatics, computer vision and recently in software engineering as well. Various surveys in the software engineering literature investigate the application of topic modelling to software engineering in general (SE) [129, 173], and into the sub-domains mining software repositories (MSR) [54] and software architecture (SA) [42]. The overall goal is to exploit automated techniques to better understand the underlying systems and processes, aid in reconstructing and improving certain parts of them, and eventually increase their quality in a cost effective manner. A large volume of literature can be found on topic modelling for SE and MSR tasks, such as concept, aspect and feature mining or location from source code, clustering similar SE artifacts, recovering traceability links among heterogeneous sets of SE artifacts/entities (e.g. source code, documentation, requirements), bug localization and

prediction, test case prioritization, evolution analysis and finally clone detection [129, 54]. The common denominator of all those approaches is the fact that there exists textual content in all those artifacts. Based on a similar observation of textual content in SE artifacts, and the fact that they might also contain architectural information, another set of approaches investigate the use of topic modelling in architecture-related tasks. The exhaustive list of activities to be supported by topic modelling in the mapping study by Bi et al. [42] includes architectural understanding, (automatic) recovery and documentation on the one hand, and architectural analysis, evaluation, and maintenance on the other hand. The authors in general emphasize the value of those activities, such as architectural understanding for distribution of responsibilities in a software system, architectural analysis for evaluating conformance in the case of a layered architecture, and so on.

All the topic modelling approaches reported in the three surveys above typically operate on a set of traditional software artifacts, notably source code and documentation. In a recent work, Perez et al. [131] observe this as well, and propose applying feature location directly on the models in model-based product families. They however use it in a very particular setting: for assessing the fitness of model fragments in a query reformulation problem using genetic algorithms. To the best of our knowledge, there are no approaches in the literature which apply topic modelling for SA-related tasks in model-driven engineering and domain-specific language ecosystems, in which we are interested in this work.

Latent Dirichlet Allocation One of the most popular topic modelling techniques, also in software engineering tasks [129, 42, 54], is Latent Dirichlet Allocation (LDA) [45]. LDA is a particular probabilistic (Bayesian) variant of topic modelling, which assumes Dirichlet prior distributions on the topics (per document, θ) and words (per topic, ϕ) and fits a generative model on the word occurrences in the corpus. Similar to the vector space model setting of SAMOS, a collection of documents is transformed into a frequency matrix. Instead of the distance and measurement (as done for clustering in SAMOS), the matrix is fed to LDA, which identifies the latent variables (topics) hidden in the data. The probability distributions θ and ϕ effectively describe the entire corpus. LDA relies on a set of hyper-parameters to be set in advance, notably n being the number of topics, α and β being the parameters of the prior Dirichlet distributions, and additional ones depending on the particular inference technique used.

While the details of the statistical inference process (e.g. computing the posterior distributions using collapsed Gibbs sampling [165] as typically used in SE-related tasks) is beyond the scope of this work, from an end-user perspective the output of LDA consists of two matrices: (given the fixed number of topics) one for the probability of each document belonging to various topics (i.e. multiple topics allowed, resulting in a kind of soft clustering), and the probability of each term belonging to various topics. The term probabilities can be manually inspected, for instance, to deduce what *"concept"* the topic actually corresponds to, while topic probabilities can be used to get the most prominent topics for the documents and identify document similarities.

The regular application of LDA as described above, requires that the number of topics is given in advance (unlike e.g. some other non-parametric variants such as Hierarchical Dirichlet Process [176]). One can either rely on domain expertise with respect to the corpus such that n is already known, or try to estimate the number using various heuristics. The latter involves running LDA with a range of candidate values, and trying to optimize certain metrics: maximize the log likelihood of the inferred model [79] or minimizing the topic density [50]. There are advanced techniques aiding or automatizing this estimation process; some notable examples within the software engineering literature include Panichella et al. [129] based on genetic algorithms, and Grant et al. [78] based on heuristics using vector similarity and source code location.

LDA has a proven track record of successful application in mining problems for natural language text documents. Yet one should be cautious while applying it, especially for other types

of artifacts. First of all, there is the non-trivial task of determining the parameters of LDA in advance (such as number of topics, as discussed above); an incorrect choice of parameters [54] and even incorrect order of input [6] can lead to non-optimal results. The authors in [129] further emphasize the difference between natural language text vs. source code, the latter of which has been recently studied and found to have a higher level of regularity than text in English [83], and claim that topic modelling for source code should be treated differently in order to get better results. For other artifacts such as models, metamodels, domain-specific languages, etc. no thorough empirical studies have been conducted regarding their nature yet.

7.9 Conclusion and Future Work

In this chapter, we have presented our approach for model analytics in an industrial context, with various analyses on ASML's MDE ecosystems. We have used and extended our model analytics framework, SAMOS, to operate on ASML's languages and models. We elaborated the domain-specific extension of SAMOS, specifically for ASML's ASOME data and control models to enable clone detection on those models. We provided extensive case studies, where we performed clone detection on ASML's models, and additionally language-level analyses ranging from cross-DSL conceptual analysis and clone detection to architectural analysis for the CARM2G ecosystem. We present our findings along with valuable feedback from the domain experts on the nature of cloning in the ecosystems, and opportunities such as refactoring to support the maintenance and quality of the ever-growing and evolving ecosystems.

Besides the wide range of analyses presented in this work, there is still a lot of room for improvements and future work. While SAMOS has many combination of settings and scopes available for model clone detection, not all these combinations were chosen for the case studies (considering the time constraints of our collaborative project with ASML). As future work we could explore different aspects of comparison using the different available settings, such as type-based and idf weighting. Furthermore, as indicated in threats to validity, advanced comparison schemes (e.g. selective ordered vs. unordered comparison for different model parts) could be integrated to improve the accuracy of our clone detection. Other directions would include the detection of patterns, e.g. design patterns (as in [180]), or anti-patterns. As an example useful application of this, one could create a pattern catalogue and find what models do not adhere to these patterns (i.e. as a potential indication of unexpected behaviour in models). Finding structural clones, especially in the control models, is another promising direction for future work. Lastly, the language analyses could be improved to overcome the limitations as addressed in the threats to validity, e.g. with more sophisticated NLP and more advanced, fine-tuned topic modelling techniques. Considering the time dimension of the languages in development, it would be also very interesting to investigate their evolution, in terms of *concept drift* [206] and cloning.

Chapter 8

Towards Distributed Model Analytics

The growing number of models and other related artifacts in model-driven engineering has recently led to the emergence of approaches and tools for analyzing and managing them on a large scale. Our framework SAMOS applies techniques inspired by information retrieval and data mining to analyze large sets of models. As the data size and analysis complexity goes up, however, further scalability is needed. In this chapter we extend SAMOS to operate on Apache Spark, a popular engine for distributed big data processing, by partitioning the data and parallelizing the comparison and analysis phase. We present preliminary studies using a cluster infrastructure and report the results for two datasets: one with 250 Ecore metamodels where we detail the performance gain with various settings, and a larger one of 7.3k metamodels with nearly one million model elements for further demonstrating scalability.

8.1 Introduction

The use of models as a basis for software engineering—whether UML models used as a basis for design and implementation, or metamodels and models in model-driven engineering (MDE)—has been exponentially growing in recent years. This is witnessed by e.g. the dramatic growth of models and related artifacts present both in open source such as the GitHub repository [101, 82] and in industrial MDE ecosystems [26]. Analogously to earlier developments in source code analytics and text mining where very high volumes of data have long emerged as a reality and a challenge, this development necessitates similar approaches for analyzing *models* at larger scales. At the same time, models inherently display more complex structure, in general being graph-structured instead of the trees with (limited) cross-links typically encountered as representations of source code and natural language. Models in turn demand efficient and scalable, even if approximate, techniques for the analysis—versus comparing them one-to-one using exact but expensive techniques (graph edit distance, similarity flooding [118], etc.). As a result, analytics becomes more complex for the setting of models, both in terms of techniques needed, and of computation effort required. Note that the requirements and potential added value for (big) data analytics in the general sense (i.e. not for models) has for long been widely recognized by the community [106, 205], and is not further elaborated in this chapter.

One way to improve the performance of model analytics, for complex analyses of large datasets, is to look at distributed settings, versus running in a sequential setting on a single machine. There recently have been a few efforts of exploiting distributed computing in the MDE community, though in a different context: for model transformations [39, 49]. In this chapter, we sketch how the existing model analytics framework SAMOS can be lifted from the latter setting to a distributed setting using the Apache Spark framework for distributed computation. In Section 8.2 we give an overview of the Apache Spark framework. Section 8.3 considers how SAMOS can be modified and extended to operate in the distributed setting for potentially higher performance, while Section 8.4 discusses initial results for two sets of Ecore metamodels from a proof of concept, i.e. a version of SAMOS lifted to the Apache Spark framework, without any specific optimizations: one dataset with 250 Ecore metamodels reporting the performance gain with various settings, and a larger one of 7.3k metamodels with nearly one million model elements for further demonstrating our scalability. Section 8.5 concludes the chapter with several indications for future work.

8.2 Background: Apache Spark

Apache Spark[1] is an open-source distributed data processing engine [202], also used for Big Data Analytics. It offers a stack of technologies for both fundamental components such as cluster management and fault-tolerant distributed data storage (in the form of Resilient Distributed Datasets — RDDs), and for advanced ones such as streaming and distributed machine learning. Spark further provides rich APIs for various programming languages including Java, Python and Scala. It improves on the previously popular MapReduce [63] computational paradigm (e.g. on Apache Hadoop[2]) with a more efficient distributed data and memory management system, leading to a higher comparative performance and scalability [202].

Spark typically operates with a master *driver* node, which coordinates several worker nodes (see Figure 8.1). Each worker node is allocated parallel tasks to process the specific parts of the distributed data (e.g. on the distributed file system). A central cache in each worker node can further improve efficiency by for instance maintaining some data in memory for faster access in multiple cores and for repeated/iterative tasks.

8.3 Distributed VSM Computation

As outlined in Chapters 2 and 3, SAMOS relies on the calculation of a vector space by comparing the extracted features of each model against the set of all features (i.e. columns of the vector space). We can identify several components of this approach:

- extraction of desired features from the models, mapping each model to the feature set P_i,
- calculation of the set of all unique features (F, dimensions of the VSM),
- given F, comparing each feature in P_i against each feature in F to calculate a row or *vector* in the VSM.

The vector represents the model in the high-dimensional vector space, to be used for distance calculation and other statistical analyses. The bottleneck of this approach is the quadratic number

[1] `https://spark.apache.org/`
[2] `http://hadoop.apache.org/`

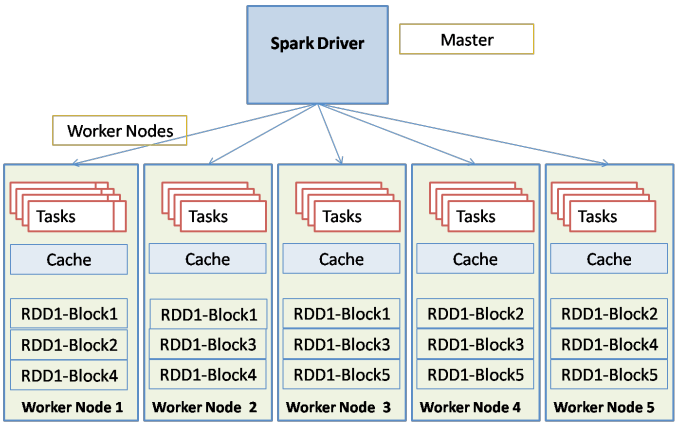


Figure 8.1: Overview of Spark architecture[3].

of feature comparisons (in contrast with the previous steps which have linear complexity), which makes this the target for our parallelization effort to increase scalability. Note that the efficiency and scalability of the distance calculation for the resulting large sparse matrix is a relatively lesser problem and left as future work.

On the other hand, thinking in the context of the Spark architecture (cf. Figure 8.1), we can map our approach to the distributed setting as follows:

- **data:** feature sets (P_i's) residing as distributed data (i.e. RDDs of model-feature pairs),
- **cache:** maximal feature set (F) precomputed and distributed to each worker node to be held in memory cache,
- **tasks:** feature comparison as the atomic unit of parallel execution.

Feature-to-feature comparison is the atomic unit for parallelization in this setting. However for practical reasons, notably to have an easier *Reduce* step of collecting back and merging the data, we aim for a coarser granularity. We perform a single pass for each feature, comparing it with the maximal set. Each parallel task in return consists of (1) pair-wise comparing a feature p in P_i versus F and computing an intermediate vector, and (2) computing the final VSM vector for the corresponding model, e.g. via summing the intermediate ones (*frequency* setting of SAMOS [19]). To exemplify, a model consisting of m features is processed m-way and eventually integrated to calculate a single row of the VSM corresponding to that model.

Note that a great deal of the necessary functionality for distributed operation is provided by Spark: partitioning and distribution (shuffling) of the data, synchronisation of the tasks and the workflow, data collection and I/O, and so on. The necessary modifications for SAMOS mostly were wrapping the related building blocks (e.g. parsing and extraction, feature comparison) into parallel Spark RDD operations with minimal glue code around them.

8.4 Preliminary Results and Discussion

We performed some preliminary experiments for our technique. First of all, we used SURFSara [4], the computational infrastructure for ICT research in the Netherlands. SURFSara provides a Hadoop cluster with Spark support, which consists of 170 data/compute nodes with 1370 CPU-cores for parallel processing and a distributed file system with a capacity of 2.3 PB.

Next, as for SAMOS, we chose bigrams of attributed nodes (for the clone detection scenario [18], see Chapter 6), as one of the more computationally intensive settings (e.g. compared with extracting simple word features for domain analysis [22]). As for the dataset, we mined GitHub for (1) a limited set of 250 Ecore[5] metamodels, and (2) a large set of 7312 Ecore metamodels (after both the exact duplicates and the files smaller than 2KB are removed). Table 8.1 shows some details on the sizes of the two datasets. A further SAMOS framework setting to mention is that we have turned off expensive NLP checks for semantic relatedness and synonymy for this preliminary experiment. For a table representation of the configuration, please refer to the bigram setting as depicted Table 6.2 in Chapter 6; the only difference being WordNet switched off.

Normally, we have a simplistic *all or none* strategy for NLP-caching; for small datasets we iterate over all the model elements to compute and keep in memory the word-to-word similarity scores (i.e. full caching). For the distributed execution we have disabled this feature as we cannot fit the relevant data for very large model sets (the goal is to process tens of thousands of models) into the memory, so we completely disabled NLP-caching. As future work, we plan to investigate various more sophisticated approaches to caching to circumvent this issue.

dataset	#models	file size	#model elem.
1	250	4.8MB	∼50k
2	7312	133.6MB	∼1 million

Table 8.1: Description of the datasets: number of metamodels, total file size and number of model elements.

Performance for Dataset 1. On dataset 1, we ran the single-core local version of SAMOS, with and without NLP-caching, and the distributed version with 1, 10, 50, 100 and 250 and 500 executors without NLP-caching. Figure 8.2 depicts the results. For the single-core case, local execution has the best performance, especially with NLP-caching enabled. We have included the single-core distributed case to roughly assess the overhead: 17.1 hours (distributed) versus 13.8 hours (local). It is evident that as the number of executors increase, the performance increases as well, though with diminishing returns.

Performance for Dataset 2. As a bigger challenge for our approach, we made an attempt to run dataset 2 with the same (expensive) settings as above. We could argue going for more approximate, hence cheaper, settings (unigrams instead of bigrams, ignoring instead of including attributes, etc.) for such a large dataset but we performed this experiment in order to load-test and assess the limits of our technique. We successfully calculated the VSM using a total of

[3] http://spideropsnet.com/site1/blog/2014/12/09/igniting-the-spark/

[4] https://www.surf.nl/en/about-surf/subsidiaries/surfsara/

[5] https://www.eclipse.org/modeling/emf/

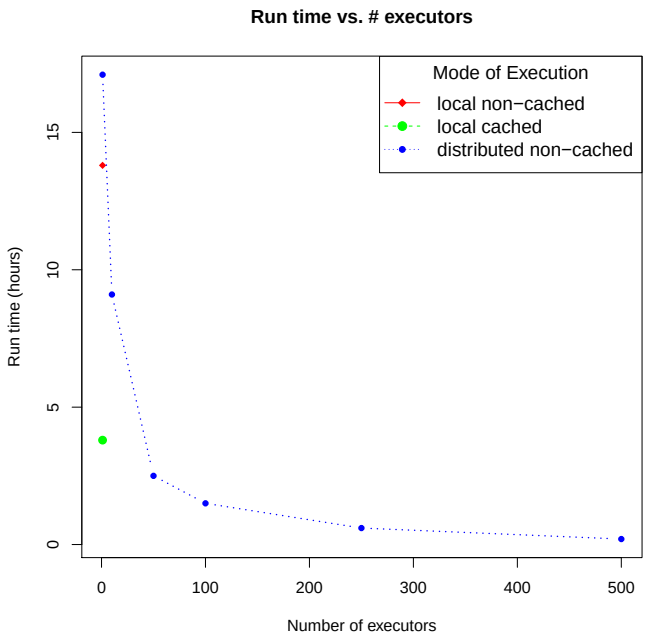


Figure 8.2: Performance for dataset 1.

∼1500 executors (215 worker nodes with 7 cores and 8GB memory each) processing the 5000-way partitioned data on SURFSara and obtained the resulting VSM (5000-part on the distributed file system) in approximately 17.9 hours.

Discussion. The results indicate that the distributed execution mode has the potential to increase the applicability of SAMOS for large datasets. We can consider this as a first but important step towards the in-depth analysis of thousands of models for application scenarios such as repository mining (notably our ∼7k Ecore metamodel set from GitHub and the ∼93k Lindholmen UML dataset [82]), large-scale clone detection and model evolution studies.

Given the preliminary nature of this work, there is certainly a lot of room for further optimization. These would involve not only optimizations with respect to the inner mechanisms of Apache Spark, but also improvements within SAMOS, which is itself a research prototype. Nevertheless, to our knowledge we do not know of another comparable model analytics approach or tool in the literature which is capable of such scalability.

Threats to Validity. In this work, we only deal with VSM calculation (which we assume to be the major bottleneck) and leave tackling the rest of the workflow—such as distance calculation and statistical analyses—as future work. We plan to proceed with parallel or scalable techniques for this as well. Another threat to validity is that our approach at this point has not been tried on larger scales, so it should be investigated how it performs with e.g. hundred thousands of models towards Big Data.

8.5 Conclusion

In this chapter we present a novel approach for distributed model analytics. We have extended the SAMOS framework to operate on Apache Spark infrastructure and exploit its powerful distributed data storage and processing facilities. Using the SURFSara cluster, we have performed preliminary experiments using two sets of Ecore metamodels. On the smaller one, we have reported in detail the performance of the different execution modes and number of executors. For the larger one, we have tested the scalability of our approach in the case of nearly a million model elements.

There is a large volume of potential future work. The immediate next step would involve extending the SAMOS workflow with scalable, distributed techniques for distance calculation and statistical analyses. More advanced statistical analyses, including predictive and prescriptive ones, are among the notable targets for future work. Noting the benefits of caching for the NLP part, we also would like to investigate various caching strategies, applicable for large amounts of data. Moreover, we believe there is a lot of room for optimization for this approach, which can be considered in parallel to the other proposed steps. A more in-depth discussion of the distributed vs. local execution in terms of performance gain and optimal number of executors would also be beneficial.

Chapter 9

Conclusions

This chapter concludes this thesis by discussing the main contributions, obstacles for model analytics and management research and directions for the future of our framework and research. For each of the research questions stated in Chapter 1, we outline the contribution with respect to the Chapters 2 to 8. Additional details are available in the individual chapters that cover the research questions.

9.1 Contributions

The starting point of this thesis was our observation that expanding MDE adoption leads to more numerous and diverse artifacts, such as models, which in turn calls for the need for automatically analyzing and managing them in a scalable manner. We elaborated this problem in Chapter 1, where we provide quantitative evidence from open repositories and industry. Based on that observation, we formulated the main research question covered in this thesis:

> **RQ:** *How can we analyze, compare and visualize large sets of models in a generic and scalable way?*

As a central architecture and customizable workflow, we introduced the SAMOS model analytics framework. The main research question is divided into three more specific research questions, and each of these questions was addressed in this thesis. The first of these questions deals with efficient model fragmentation to enable large scale analysis of models.

> **RQ$_1$:** *How (i.e. with which features) can we represent models and their relevant information for scalable analyses?*

To address this question, we presented an approach inspired by the domain of information retrieval. In Chapter 2, we outlined the basic idea. We extracted simplistic features from models in the form of type-name pairs, which correspond to the relevant vertex information in the underlying graphs of the models. Realizing the need for more advanced and detailed fragments (i.e. more accurate representations of the models), we further introduced the following features:

n-grams (Chapter 3) and subtrees (Chapter 6) which capture structural information, attributed nodes (Chapter 6) to also capture additional vertex information such as modifiers and sets (Chapter 5) to represent unordered collections. We further used a very simplistic (therefore cheap) preprocessed name-only representation (e.g. in Chapter 6), as a means of highly efficient way of dealing with tens of thousands of models in certain scenarios. This rich set of representations effectively allows SAMOS to fragment various model types as needed.

> **RQ_2:** *What techniques can we use to compare models (represented as fragments) in order to discover e.g. similarities, clusters and outliers?*

This question maps to the feature comparison and clustering steps of the SAMOS workflow. We devised a VSM-based approach for model analytics, where we compute the VSM via several feature comparison and weighting settings. For comparing features, we applied a range of techniques: natural language processing (including tokenization, stop word filtering and checking semantic relatedness) for textual content (Chapter 2), maximum similar subsequence for n-grams (Chapter 3), tree edit distance and modified Hungarian algorithm for trees (Chapter 6) and again modified Hungarian algorithm for unordered sets (Chapter 5). We further improved and fine-tuned the VSM by idf-based and type-based weighting schemes (Chapter 2).

Using the computed VSM, we used various distance measures: cosine (Chapter 2) for angular similarity and masked Bray-Curtis (Chapter 6) for normalized similarity. Depending on the nature of the problem at hand, we applied different analyses, such as hierarchical clustering for visualization (Chapter 2), density-based clustering for clone detection (Chapter 6) and topic modelling (Chapter 7) for architectural recovery.

Moreover, we further experimented with a distributed data processing back-end for SAMOS (Chapter 8), as a means to improve its scalability further, to help with e.g. larger datasets or more expensive analyses.

As for the third research question, a wide range of analysis techniques integrated in SAMOS allowed us to explore different application areas for our approach:

> **RQ_3:** *How can we exploit the information we acquire through these large scale model analyses, in order to improve the state-of-the-art MDE practices for model analytics and management?*

Throughout the thesis, we used our approach for a variety of application areas around the topic of large scale model analytics and management. In Chapter 2, we probed into how the problems of model searching, domain analysis, model repository exploration and management (for Ecore metamodels in particular) could benefit from our technique. In a variability mining scenario (Chapter 4), we applied SAMOS as a preprocessing step to improve the quality of the inferred variability models from state chart variants. We further applied SAMOS in a repository management setting for feature models in Chapter 5 in terms of domain analysis and clone detection. We advanced and polished the clone detection capabilities of SAMOS, and comparatively evaluated it against state-of-the-art model clone detectors in Chapter 6. There we also investigated the application of SAMOS for detecting clones in DSLs (i.e. with respect to their metamodels) in GitHub. Finally we performed a wide range of studies in industrial MDE ecosystems: clone detection in industrial data and control models, cross-ecosystem and cross-DSL comparison and clone detection, and architectural reconstruction and analysis of an MDE ecosystem (Chapter 7). Overall we demonstrate how model analytics can be beneficial in the context of model management, in terms of e.g. model searching, repository management, model maintenance and reuse.

As can be followed from the explicit set of configurations reported in each chapter, we continuously experimented and improved the components and settings of SAMOS. This reflects our thinking process and growing insight into our technique and its application to a variety of scenarios.

9.2 Obstacles for Model Analytics and Management

Throughout our research we attacked some of the problems in the MDE domain using cross-disciplinary techniques. We had a considerable amount of success in doing so, as evaluated in a range of settings. The workflow inspired by IR eventually can indeed be applied to models provided that the necessary extensions are integrated. The clustering (i.e. unsupervised machine learning) mindset of finding groups of similar models and outliers has also led to a number of successful application areas around model management. Nevertheless, we would like to explicitly address some of the limitations we encountered.

- *Analyzing Graph Data.* The techniques which we adopted and experimented with are mostly from the text mining and information retrieval domains as well as data science for simple types of data. This is, for instance, the reason why n-grams are generally acceptable for approximating the structure in natural language text, yet were found to be inadequate in our clone detection studies (see Chapter 5 for the discussion with the n-ary relation in the form of group cardinalities in feature models, Chapter 6 for the inaccuracies in general). It is an obstacle and an open question for model analytics in general, how to exploit the relevant techniques, including natively graph-based approaches (e.g. graph kernels) or other advanced techniques (e.g. locality-sensitive hashing on graph data). We however would require them to be (1) accurate yet reasonably scalable, and (2) applicable to the problems at hand in the MDE domain.

- *Processing Textual Content in Models.* We used a range of NLP techniques in our approach, to process the textual content in models (e.g. in model element names). We found it very difficult to do this accurately, e.g. compared to the traditional NLP tasks. First of all, we might not have *big enough* data (i.e. models) in the MDE domain to do an unsupervised approach for synonym detection, e.g. via word embeddings. Another limitation is that the textual content in the models is typically short, partially cryptic, domain-specific and lack proper sentential context. To exemplify one of the pains, we had very little success in our exploratory efforts in trying to do a proper word-sense disambiguation using the surrounding model elements in the underlying graph as context (using variations of the Lesk algorithm [107]).

- *Lack of Labeled Data for Model Analytics.* We mostly had to use unsupervised techniques supported with manual validation in our work. This is mostly due to the fact that there is a lack of structured and labelled datasets in our domain. This has a stark contrast with e.g. text mining communities which have been building and studying many standard datasets for decades. In the future, we hope to contribute to the MDE research community in this respect as well; via student projects or crowd-sourced annotations (e.g. Amazon Mechanical Turk[1]). Once we have a labelled training set, it is quite possible to *plug in* more advanced machine learning algorithms and explore applications of prescriptive/predictive statistical techniques and supervised approaches.

[1] `https://www.mturk.com/`

- *Lack of Awareness in the MDE Community.* In our numerous discussions with MDE researchers and practitioners, we observed that an important part of the community is not aware of the rapidly increasing scale of MDE artifacts along with increasingly widespread MDE practices. This has been recently addressed by van den Brand [183] in an industrial context as well. To give some anecdotal evidence, we have so far received some interesting feedback ranging from *"There aren't so many models in the whole world combined, why are you making up an artificial problem?"* and *"If there are multiple DSLs and metamodels, this means the language engineers in the company are doing their job wrong."* to *"Machine learning on models doesn't work, believe me I've tried everything!"* and *"Model clone detection research is dead!"*. Note that this perception was much stronger a few years ago; the awareness seems to slowly be increasing recently, partly thanks to the seminal work by Hebig et al. with the Lindholmen UML dataset [82], our dissemination efforts via workshops, and emerging literature on various related topics (see Chapter 1 for examples). However, we believe it will take some more time until model analytics and management (as in our vision) will settle as an established sub-domain within MDE.

9.3 Future Work

In this thesis we presented a generic model analytics and management approach, applied in a variety of scenarios. However, we believe there is much more potential for our technique. A short summary of those, i.e. future directions for our work would include the following:

- Extending SAMOS for different model types, notably UML models, Simulink models and business process models, each with its own difficulties and domain-specific concerns to tackle;
- Extending SAMOS for different type of analyses, notably for detecting structural clones, semantic clones, and patterns in the general sense;
- Enabling model analytics and management for heterogeneous sets of MDE artifacts, including e.g. different UML notations (concrete syntaxes as well as UML 1.x vs. 2.x), different domain model notations (e.g. UML vs. Ecore), and models from multiple disciplines (software, hardware, engineering, business processes);
- Incorporating new features such as metrics, constraints and subgraphs for extraction and comparison;
- Improving the NLP part with advanced features such as part-of-speech tagging, word sense disambiguation, domain specific dictionaries and even cross-language capabilities;
- Exploring different problems such as model classification, model querying and searching and MDE (co-)evolution;
- Experimenting with potentially more powerful settings for SAMOS, such as advanced weighting schemes, flexible scoping, distance measures and clustering algorithms, and eventually other fundamental approaches beyond VSM such as scalable graph-based approaches;
- Experimenting with other statistical techniques beyond descriptive ones; predictive and prescriptive ones, supervised machine learning and data mining techniques; and
- Maturing and fully integrating the distributed processing back-end to maximize the scalability.

9.4 The Future of SAMOS as a Mature Open Framework

Orthogonal to these directions, we have put some initial effort in transforming SAMOS from a research prototype into a mature, open, extensible and usable framework for model analytics. We believe this to be an important aspect for the longevity of SAMOS as a state-of-the-art tool.

We opted for integrating SAMOS into the open source Konstanz Information Miner (KNIME) data mining framework [41]. Building on the Eclipse ecosystem[2] (which fits perfectly with the existing Eclipse-based MDE toolset such as EMF), KNIME has a flexible plug-in architecture and allows easy visual assembly and interactive execution of a data pipeline. It already comes with a plethora of modules implementing techniques from e.g. data mining and machine learning. Moreover, thanks to its modular environment, one can easily integrate new algorithms, tools, data manipulation and visualization capabilities.

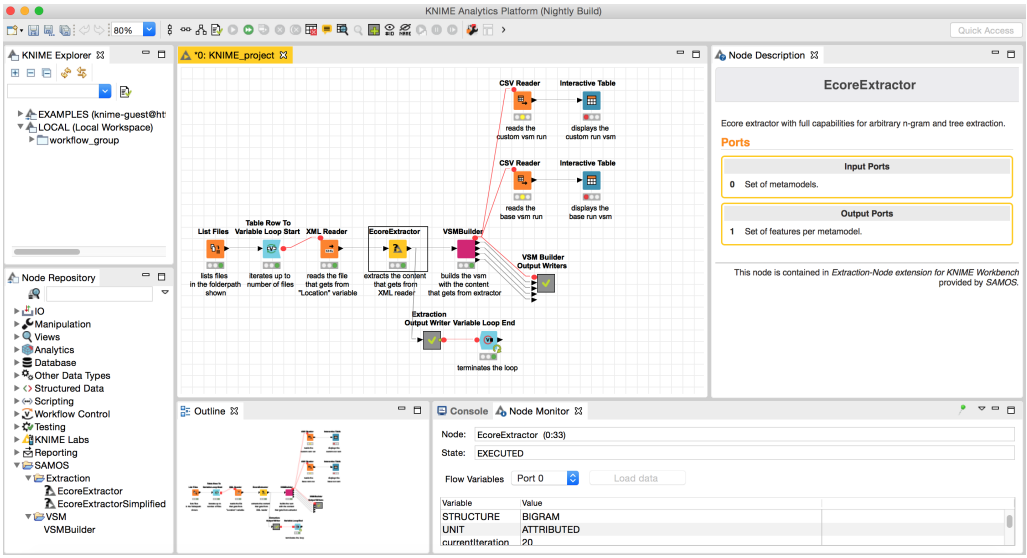


Figure 9.1: Describing SAMOS workflows visually in KNIME.

Figure 9.1 depicts a screenshot of the version of SAMOS running on KNIME. Here is a summary of the major components:

- On the lower left side, there is the node repository containing all the components already present in KNIME, and additionally the SAMOS-specific extensions: an extraction category and a VSM category, containing the *EcoreExtractor, EcoreExtractorSimplified* and *VSMBuilder* nodes.
- Using the nodes, we can design a visual model analytics workflow by drag-and-dropping them into the center part, i.e. the workflow window. The nodes have well-defined input and output data ports. A short description of their functionality along with their ports can be found in the right-most Node Description window (upon selecting the desired node).
- Workflows can be easily stored, modified and reused. The left-top window, i.e. KNIME Explorer shows the various user-defined workflows in the workspace.

[2] http://www.eclipse.com/

- KNIME allows real-time monitoring of the workflow progress. Each node in the workflow is shown with red, yellow or green lights (and flashing lights) to indicate the status e.g. whether they are still running, or have successfully completed. The lower right side shows detailed information about the status of each node.
- We get, for free, a simple and convenient GUI for setting the custom parameters for each node. This is an improvement over the originally file-based configuration of SAMOS, and is actually a feature that was explicitly asked from by industrial partners to increase our tool's usability. Figure 9.2 shows for instance the parameters of the scope, unit, structure and strategy for feature extraction, along with preprocessing flags.

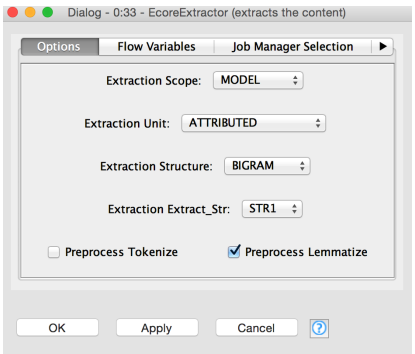


Figure 9.2: A simple GUI for setting up the parameters of SAMOS feature extraction node.

The workflow in Figure 9.1 contains, among others, file I/O components which are used directly from the existing set of KNIME nodes. We believe such opportunities for reusing existing functionality would save us the effort for developing standard functionality from scratch in SAMOS in the future. We have also experimented with, and plan to use in the near future e.g. visual descriptive statistics nodes of KNIME to aid our empirical analyses on MDE artifacts, and machine learning components to build intelligent predictive models for various MDE tasks.

Bibliography

[1] S. L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *Int. Conf. on Program Comprehension*, pages 156–159. IEEE, 2010.

[2] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature model differences. In *Int. Conf. on Advanced Information Systems Engineering*, pages 629–645. Springer, 2012.

[3] M. Acher, B. Baudry, P. Heymans, A. Cleve, and J.-L. Hainaut. Support for reverse engineering and maintaining feature models. In *Int. Workshop on Variability modelling of Software-intensive Systems*, page 20. ACM, 2013.

[4] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. B. France. Composing your compositions of variability models. In *Int. Conf. on Model Driven Engineering Languages and Systems*, pages 352–369. Springer, 2013.

[5] S. Adyanthaya. *Robust multiprocessor scheduling of industrial-scale mechatronic control systems*. PhD thesis, Eindhoven: Technische Universiteit Eindhoven, 2016.

[6] A. Agrawal, W. Fu, and T. Menzies. What is wrong with topic modeling? and how to fix it using search-based software engineering. *Information and Software Technology*, 98: 74–88, 2018.

[7] B. Al-Batran, B. Schätz, and B. Hummel. Semantic Clone Detection for Model-Based Development of Embedded Systems. In *Int. Conf. on Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 258–272. Springer, 2011. ISBN 978-3-642-24484-1.

[8] M. Alalfi, E. Rapos, A. Stevenson, M. Stephan, T. Dean, and J. Cordy. Semi-automatic Identification and Representation of Subsystem Variability in Simulink Models. In *Int. Conf. on Software Maintenance and Evolution*, pages 486–490. IEEE, 2014.

[9] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for simulink models. In *Int. Conf. on Software Maintenance*, pages 295–304. IEEE, 2012.

[10] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Analysis and clustering of model clones: An automotive industrial experience. In *Int. Conf. on Software Maintenance, Reengineering and Reverse Engineering*, pages 375–378. IEEE, 2014.

[11] M. Alanen and I. Porres. Difference and Union of Models. In *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003. ISBN 978-3-540-20243-1.

[12] W. Alberts. ASML's MDE Going Sirius. `https://www.slideshare.net/Obeo_corp/siriuscon2016-asmls-mde-going-sirius`, 2016. Accessed: 2018-11-12.

[13] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[14] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 67–76. IEEE, 2008.

[15] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Companion Proc. of the 36th Int. Conf. on Software Engineering*, pages 532–535. ACM, 2014. ISBN 978-1-4503-2768-8.

[16] E. P. Antony, M. H. Alalfi, and J. R. Cordy. An approach to clone detection in behavioural models. In *Working Conf. on Reverse Engineering*, pages 472–476, Oct 2013.

[17] Ö. Babur. Statistical analysis of large sets of models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 888–891, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5.

[18] Ö. Babur. Clone detection for Ecore metamodels using n-grams. In *Proc. of the 6th Int. Conf. on Model-Driven Engineering and Software Development, 2018*, pages 411–419, 2018.

[19] Ö. Babur and L. Cleophas. Using n-grams for the automated clustering of structural models. In *43rd Int. Conf. on Current Trends in Theory and Practice of Computer Science*, pages 510–524, 2017.

[20] Ö. Babur, V. Smilauer, T. Verhoeff, and M. van den Brand. Multiphysics and multiscale software frameworks: An annotated bibliography. Technical Report 15-01, Dept. of Mathematics and Computer Science, Technische Universiteit Eindhoven, Eindhoven, 2015.

[21] Ö. Babur, V. Smilauer, T. Verhoeff, and M. van den Brand. A survey of open source multiphysics frameworks in engineering. *Procedia Computer Science*, 51:1088–1097, 2015.

[22] Ö. Babur, L. Cleophas, and M. van den Brand. Hierarchical clustering of metamodels for comparative analysis and visualization. In *Proc. of the 12th European Conf. on Modelling Foundations and Applications, 2016*, pages 2–18, 2016.

[23] Ö. Babur, L. Cleophas, T. Verhoeff, and M. van den Brand. Towards statistical comparison and analysis of models. In *Proceedings of the 4th Int. Conf. on Model-Driven Engineering and Software Development*, pages 361–367, 2016.

[24] Ö. Babur, L. Cleophas, and M. van den Brand. Model analytics for feature models: case studies for S.P.L.O.T. repository. In *Proc. of MODELS 2018 Workshops, co-located with ACM/IEEE 21st Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018.*, pages 787–792, 2018.

[25] Ö. Babur, L. Cleophas, and M. van den Brand. Towards distributed model analytics with Apache Spark. In *Proc. of the 6th Int. Conf. on Model-Driven Engineering and Software Development, 2018*, pages 767–772, 2018.

[26] Ö. Babur, L. Cleophas, M. van den Brand, B. Tekinerdogan, and M. Aksit. Models, more models, and then a lot more. In M. Seidl and S. Zschaler, editors, *Software Technologies: Applications and Foundations*, pages 129–135, Cham, 2018. Springer International Publishing. ISBN 978-3-319-74730-9.

[27] Ö. Babur, L. Cleophas, and M. van den Brand. Metamodel clone detection with SAMOS. *Journal of Visual Languages and Computing (JVLC)*, accepted for publication.

[28] Ö. Babur, L. Cleophas, and M. van den Brand. Metamodel clone detection with SAMOS (extended abstract). In *The 17th edition of the BElgian-NEtherlands software eVOLution symposium (BENEVOL), 2018*, accepted for publication and presentation.

[29] Ö. Babur, A. Suresh, W. Alberts, L. Cleophas, R. Schiffelers, and M. van den Brand. Model analytics for industrial MDE ecosystems. In B. Tekinerdogan, Ö. Babur, L. Cleophas, M. van den Brand, and M. Aksit, editors, *Model Management and Analytics for Large Scale Systems*. Elsevier, submitted as a book chapter.

[30] I. Baki and H. A. Sahraoui. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Trans. Softw. Eng. Methodol.*, 25(3):20:1–20:37, 2016. doi:10.1145/2904904. URL `http://doi.acm.org/10.1145/2904904`.

[31] A. Barriga, A. Rutle, and R. Heldal. Automatic model repair using reinforcement learning. In *Proc. of MODELS 2018 Workshops, co-located with ACM/IEEE 21st Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018.*, pages 781–786, 2018.

[32] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio. MDEForge: an extensible web-based modeling platform. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS 2014, Valencia, Spain, September 30, 2014.*, pages 66–75, 2014. URL `http://ceur-ws.org/Vol-1242/paper10.pdf`.

[33] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. A tool for clustering metamodel repositories. In *Demonstrations and Posters at MODELS2015, Ottawa, Canada*, 2015.

[34] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Model repositories: Will they become reality? In *CloudMDE@ MoDELS*, pages 37–42, 2015.

[35] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Automated clustering of metamodel repositories. In *Int. Conf. on Advanced Information Systems Engineering*, pages 342–358. Springer, 2016.

[36] G. Bécan, S. Ben Nasr, M. Acher, and B. Baudry. WebFML: synthesizing feature models everywhere. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*, pages 112–116. ACM, 2014.

[37] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[38] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, a scalable persistence layer for EMF models. In *European Conference on Modelling Foundations and Applications*, pages 230–241. Springer, 2014.

[39] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot. Distributed model-to-model transformation with ATL on MapReduce. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 37–48. ACM, 2015.

[40] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39–48. IEEE, 2000.

[41] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel. KNIME-the Konstanz information miner: version 2.0 and beyond. *ACM SIGKDD explorations Newsletter*, 11(1):26–31, 2009.

[42] T. Bi, P. Liang, A. Tang, and C. Yang. A systematic mapping study on text analysis techniques in software architecture. *Journal of Systems and Software*, 144:533–558, 2018.

[43] B. Bislimovska, A. Bozzon, M. Brambilla, and P. Fraternali. Textual and content-based search in repositories of web application models. *ACM Transactions on the Web (TWEB)*, 8(2):11, 2014.

[44] D. M. Blei and J. D. Lafferty. Topic models. In *Text Mining*, pages 101–124. Chapman and Hall/CRC, 2009.

[45] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[46] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012. ISBN 1608458822, 9781608458820.

[47] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Proc. of the 2006 Int. Workshop on Global Integrated Model Management*, pages 5–12. ACM, 2006.

[48] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, pages 1–47, 2015.

[49] L. Burgueño, M. Wimmer, and A. Vallecillo. Towards distributed model transformations with LinTra. 2016.

[50] J. Cao, T. Xia, J. Li, Y. Zhang, and S. Tang. A density-based method for adaptive LDA model selection. *Neurocomputing*, 72(7):1775 – 1781, 2009. Advances in Machine Learning and Computational Intelligence.

[51] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[52] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data*, pages 493–504. ACM, 1996.

[53] J. Chen, T. R. Dean, and M. H. Alalfi. Clone detection in MATLAB stateflow models. *Software Quality Journal*, 24(4):917–946, 2016.

[54] T.-H. Chen, S. W. Thomas, and A. E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016.

[55] R. Clarisó and J. Cabot. Applying graph kernels to model-driven engineering problems. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, MASES 2018, pages 1–5, 2018.

[56] P. C. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN 0-201-70332-7.

[57] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, and A. Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018.

[58] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[59] J. R. Cordy and C. K. Roy. The NiCad clone detector. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 219–220. IEEE, 2011.

[60] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[61] G. Daniel, G. Sunyé, and J. Cabot. Mogwaï: a framework to handle complex queries on large models. In *Research Challenges in Information Science (RCIS), 2016 IEEE Tenth International Conference on*, pages 1–12. IEEE, 2016.

[62] L. De Alfaro, M. Faella, and M. Stoelinga. Linear and branching metrics for quantitative transition systems. In *International Colloquium on Automata, Languages, and Programming*, pages 97–109. Springer, 2004.

[63] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[64] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 603–612. IEEE, 2008.

[65] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *Proc. of the 4th Int. Workshop on Software Clones*, pages 57–64. ACM, 2010.

[66] M. M. Deza and E. Deza. *Encyclopedia of Distances*. Springer, 2009.

[67] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Mining metrics for understanding metamodel characteristics. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, MiSE 2014, pages 55–60, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2849-4. doi:10.1145/2593770.2593774. URL http://doi.acm.org/10.1145/2593770.2593774.

[68] DiffPlug Simulink. https://www.diffplug.com/features/simulink.

[69] R. Dijkman, M. Dumas, B. van Dongen, R. Käärik, and J. Mendling. Similarity of business process models: Metrics and evaluation. *Inf. Systems*, 36(2):498–516, 2011.

[70] N. Dintzner, A. van Deursen, and M. Pinzger. Analysing the linux kernel feature model changes using FMDiff. *Software & Systems Modeling*, pages 1–22, 2015.

[71] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *17th European Conf. on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013. ISBN 978-0-7695-4948-4.

[72] C. C. Ekanayake, M. Dumas, L. García-Bañuelos, M. La Rosa, and A. H. ter Hofstede. Approximate clone detection in repositories of business process models. In *International Conference on Business Process Management*, pages 302–318. Springer, 2012.

[73] EnSoft SimDiff. http://www.ensoftcorp.com/simdiff/.

[74] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the Second Int. Conf. on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[75] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. Building Software Product Lines from Conceptualized Model Patterns. In *Proc. of the 19th Int. Conf. on Software Product Line*, pages 46–55. ACM, 2015. ISBN 978-1-4503-3613-0.

[76] J. Font, M. Ballarín, Ø. Haugen, and C. Cetina. Automating the Variability Formalization of a Model Family by Means of Common Variability Language. In *Proc. of the 19th Int. Conf. on Software Product Line*, pages 411–418. ACM, 2015. ISBN 978-1-4503-3613-0.

[77] H. Frank and J. Eder. Towards an Automatic Integration of Statecharts. In *Proc. of the 18th Int. Conf. on Conceptual Modeling*, volume 1728 of *LNCS*, pages 430–445. Springer, 1999. ISBN 978-3-540-66686-8.

[78] S. Grant and J. R. Cordy. Estimating the optimal number of latent concepts in source code analysis. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 65–74. IEEE, 2010.

[79] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National academy of Sciences*, 101(suppl 1):5228–5235, 2004.

[80] T. Hartmann, A. Moawad, F. Fouquet, G. Nain, J. Klein, Y. L. Traon, and J.-M. Jezequel. Model-driven analytics: Connecting data, domain knowledge, and learning. *arXiv preprint arXiv:1704.01320*, 2017.

[81] A. Hashibon, Ö. Babur, M. Hanzich, G. Houzeaux, and B. Patzák. *Platforms for ICME*, chapter 8, pages 533–564. Wiley-Blackwell, 2016. ISBN 9783527693566.

[82] R. Hebig, T. H. Quang, M. R. Chaudron, G. Robles, and M. A. Fernandez. The quest for open source projects that use UML: mining GitHub. In *Proc. of MODELS '19*, pages 173–183. ACM, 2016.

[83] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[84] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. Family model mining for function block diagrams in automation software. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*, pages 36–43. ACM, 2014.

[85] A. Hotho, A. Nürnberger, and G. Paass. A brief survey of text mining. *LDV Forum*, 20(1):19–62, 2005. URL `http://www.jlcl.org/2005_Heft1/19-62_HothoNuernbergerPaass.pdf`.

[86] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 377–386. IEEE Press, 2013.

[87] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9. IEEE, 2010.

[88] IBM Rational Rhapsody. `http://www.ibm.com/software/awdtools/rhapsody/`.

[89] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[90] F. Javed, M. Mernik, J. Gray, and B. R. Bryant. Mars: A metamodel recovery system using grammar inference. *Inf. and Software Tech.*, 50(9):948–968, 2008.

[91] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, Mar. 2007. ISSN 1532-060X. doi:10.1002/smr.344. URL `http://dx.doi.org/10.1002/smr.344`.

[92] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[93] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[94] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645, 2008.

[95] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. Understanding Model Evolution through Semantically Lifting Model Differences with SiLift. In *28th Int. Conf. on Software Maintenance*, pages 638–641. IEEE, 2012.

[96] U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering*, volume 64 of *LNI*, pages 105–116. GI, 2005.

[97] B. Klatt, M. Küster, and K. Krogmann. A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies. In *1st Int. Workshop on Reverse Variability Engineering*, pages 1–8. ACM, 2013.

[98] P. Klint, D. Landman, and J. Vinju. Exploring the limits of domain model recovery. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 120–129. IEEE, 2013.

[99] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. ICSE Workshop on*, pages 1–6. IEEE, 2009.

[100] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2165-5. doi:10.1145/2487766.2487768.

[101] D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige. Assessing the use of Eclipse MDE technologies in open-source software projects. In *OSS4MDE@ MoDELS*, pages 20–29, 2015.

[102] T. Kosar, S. Bohra, and M. Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77 – 91, 2016. ISSN 0950-5849.

[103] R. Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[104] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[105] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[106] S. LaValle, E. Lesser, R. Shockley, M. S. Hopkins, and N. Kruschwitz. Big data, analytics and the path from insights to value. *MIT sloan management review*, 52(2):21, 2011.

[107] M. Lesk. Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *Proceedings of the 5th annual international conference on Systems documentation*, pages 24–26. ACM, 1986.

[108] Z. Liang, Y. Cheng, and J. Chen. A Novel Optimized Path-Based Algorithm for Model Clone Detection. *Journal of Software*, 9(7):1810–1817, 2014.

[109] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study. Technical Report 2012-07, Technische Universität Braunschweig, Germany, 2012.

[110] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *Software Engineering Conf., 2006. APSEC 2006. 13th Asia Pacific*, pages 269–276. IEEE, 2006.

[111] D. Lucrédio, R. P. d. M. Fortes, and J. Whittle. Moogle: a metamodel-based model search engine. *Software & Systems Modeling*, 11(2):183–208, 2012.

[112] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

[113] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[114] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

[115] J. Martinez, T. Ziadi, J. Klein, and Y. Le Traon. Identifying and Visualising Commonality and Variability in Model Variants. In *European Conf. on Modelling Foundations and Applications*, volume 8569 of *LNCS*, pages 117–131. Springer, 2014. ISBN 978-3-319-09194-5.

[116] Y. Mass and M. Mandelbrod. Retrieving the most relevant xml components. In *INEX 2003 Workshop Proceedings*, page 58. Citeseer, 2003.

[117] A. Mehra, J. Grundy, and J. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Prof. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 204–213. ACM, 2005. ISBN 1-58113-993-4.

[118] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Data Engineering, 2002. Proc. 18th Int. Conf. on*, pages 117–128. IEEE, 2002.

[119] M. Mendonca, M. Branco, and D. Cowan. SPLOT: software product lines online tools. In *Proc. of the 24th ACM SIGPLAN Conf. Companion on Object oriented Prog. Systems Languages and Applications*, pages 761–762. ACM, 2009.

[120] J. Mengerink, A. Serebrenik, R. Schiffelers, and M. van den Brand. Automated analyses of model-driven artifacts: obtaining insights into industrial application of MDE. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, pages 116–121. ACM, 2017.

[121] J. G. M. Mengerink, J. Noten, and A. Serebrenik. Empowering ocl research: a large-scale corpus of open-source data from github. *Empirical Software Engineering*, 2018.

[122] M. Mondai, C. K. Roy, and K. A. Schneider. Micro-clones in evolving software. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 50–60. IEEE, 2018.

[123] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *Software Engineering, 2007. ICSE 2007. 29th Int. Conf. on*, pages 54–64. IEEE, 2007.

[124] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Variant Feature Specifications. *Transactions on Software Engineering*, 38(6):1355–1375, 2012.

[125] J. Nogueira Bastos. *Modular specification and design exploration for flexible manufacturing systems*. PhD thesis, Department of Electrical Engineering, 12 2018. Proefschrift.

[126] F. N. A. A. Omran and C. Treude. Choosing an NLP library for analyzing software documentation: a systematic literature review and a series of experiments. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 187–197, 2017. doi:10.1109/MSR.2017.42. URL https://doi.org/10.1109/MSR.2017.42.

[127] M. H. Osman, T. Ho-Quang, and M. Chaudron. An automated approach for classifying reverse-engineered and forward-engineered UML class diagrams. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 396–399. IEEE, 2018.

[128] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Int. Conf. on Software Product Line*, pages 6:1–6:8. ACM, 2011. ISBN 978-1-4503-0789-5.

[129] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 522–531. IEEE Press, 2013.

[130] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[131] F. Pérez, J. Font, L. Arcega, and C. Cetina. Automatic query reformulations for feature location in a model-based family of software products. *Data & Knowledge Engineering*, 116:159 – 176, 2018. ISSN 0169-023X.

[132] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st Int. Conf. on Software Engineering*, pages 276–286. IEEE Computer Society, 2009.

[133] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. ISBN 3540243720.

[134] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *J. UCS*, 8(11):1016, 2002.

[135] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL http://www.R-project.org/.

[136] J. A. Ramey. *clusteval: Evaluation of Clustering Algorithms*, 2012. URL http://CRAN.R-project.org/package=clusteval. R package version 0.1.

[137] D. Ratiu, M. Feilkas, and J. Jürjens. Extracting domain ontologies from domain specific apis. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conf. on*, pages 203–212. IEEE, 2008.

[138] D. Rattan, R. Bhatia, and M. Singh. Model clone detection based on tree comparison. In *2012 Annual IEEE India Conference (INDICON)*, pages 1041–1046, Dec 2012.

[139] I. Reinhartz-Berger. Towards automatization of domain modeling. *Data & Knowledge Engineering*, 69(5):491–515, 2010.

[140] C. K. Roy. Detection and analysis of near-miss software clones. In *Ph. D. Thesis, Queen's School of Computing*. Citeseer, 2009.

[141] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical Report 541, School of Computing, Queen's University, Kingston, Ontario, Canada, 2007.

[142] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.

[143] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009. ISSN 0167-6423. doi:http://dx.doi.org/10.1016/j.scico.2009.02.007. URL `http://www.sciencedirect.com/science/article/pii/S0167642309000367`.

[144] J. Rubin and M. Chechik. Combining Related Products into Product Lines. In *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 285–300. Springer, 2012. ISBN 978-3-642-28871-5.

[145] J. Rubin and M. Chechik. *Domain Engineering: Product Lines, Languages, and Conceptual Models*, chapter A Survey of Feature Location Techniques, pages 29–58. Springer, 2013. ISBN 978-3-642-36654-3.

[146] J. Rubin and M. Chechik. Quality of Merge-Refactorings for Product Lines. In *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 7793 of *LNCS*, pages 83–98. Springer, 2013. ISBN 978-3-642-37056-4.

[147] J. Rubin and M. Chechik. N-way model merging. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 301–311. ACM, 2013.

[148] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[149] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic Variation-point Identification in Function-block-based Models. In *Int. Conf. on Generative Programming and Component Engineering*, pages 23–32. ACM, 2010.

[150] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Extraction of Feature Models from Formal Contexts. In *Int. Conf. on Software Product Line*, pages 4:1–4:8. ACM, 2011. ISBN 978-1-4503-0789-5.

[151] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic library migration for the generation of hardware-in-the-loop models. *Sci. Comput. Programming*, 77(2):83–95, 2012.

[152] M. Sabetzadeh and S. Easterbrook. Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach. In *Int. Conf. on Automated Software Engineering*, pages 12–21. IEEE, 2003.

[153] R. Schiffelers. Empowering high tech systems engineering using mdse ecosystems (invited talk). In E. Guerra and M. van den Brand, editors, *Theory and Practice of Model Transformation - 10th International Conference, ICMT 2017 Held as Part of STAF 2017, Proceedings*, Lecture Notes in Computer Science, page XI, Germany, 2017. Springer.

[154] R. R. H. Schiffelers, W. Alberts, and J. P. M. Voeten. Model-based specification, analysis and synthesis of servo controllers for lithoscanners. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, MPM '12, pages 55–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1805-1. doi:10.1145/2508443.2508453. URL `http://doi.acm.org/10.1145/2508443.2508453`.

[155] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer. Detecting variability in matlab/simulink models: an industry-inspired technique and its evaluation. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 215–224. ACM, 2017.

[156] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[157] C. Seidl, T. Winkelmann, and I. Schaefer. A software product line of feature modeling notations and cross-tree constraint languages. *Modellierung*, 2016.

[158] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 461–470. IEEE, 2011.

[159] M. Stephan and J. R. Cordy. A survey of model comparison approaches and applications. In *Modelsward*, pages 265–277, 2013.

[160] M. Stephan and J. R. Cordy. Model clone detector evaluation using mutation analysis. In *ICSME*, pages 633–638, 2014.

[161] M. Stephan and J. R. Cordy. Identifying instances of model design patterns and antipatterns using model clone detection. In *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*, MiSE '15, pages 48–53, Piscataway, NJ, USA, 2015. IEEE Press. URL `http://dl.acm.org/citation.cfm?id=2820489.2820501`.

[162] M. Stephan and J. R. Cordy. MuMonDE: A framework for evaluating model clone detectors using model mutation analysis. *Software Testing, Verification and Reliability*, page e1669, 2018.

[163] M. Stephan, M. H. Alafi, A. Stevenson, and J. R. Cordy. Towards qualitative comparison of simulink model clone detection approaches. In *Proceedings of the 6th International Workshop on Software Clones*, pages 84–85. IEEE Press, 2012.

[164] M. Stephan, M. H. Alalfi, and J. R. Cordy. Towards a taxonomy for simulink model mutations. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 206–215. IEEE, 2014.

[165] M. Steyvers and T. Griffiths. Probabilistic topic models. *Handbook of latent semantic analysis*, 427(7):424–440, 2007.

[166] K. Stol, P. Ralph, and B. Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 120–131, May 2016.

[167] H. Störrle. Towards clone detection in UML domain models. *Software & Systems Modeling*, 12(2):307–329, 2013.

[168] H. Störrle. Effective and efficient model clone detection. In *Software, Services, and Systems*, pages 440–457. Springer, 2015.

[169] H. Störrle, R. Hebig, and A. Knapp. An index for software engineering models. In *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 28 - October 3, 2014.*, pages 36–40, 2014. URL `http://ceur-ws.org/Vol-1258/poster8.pdf`.

[170] D. Strüber, M. Selter, and G. Taentzer. Tool support for clustering large meta-models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 7. ACM, 2013.

[171] D. Strüber, J. Plöger, and V. Acreţoaie. Clone detection for graph-based model transformation languages. In *Int. Conf. on Theory and Practice of Model Transformations*, pages 191–206. Springer, 2016.

[172] D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger. RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules. In *Int. Conf. on Fundamental Approaches to Software Engineering*, pages 122–140. Springer, 2016.

[173] X. Sun, X. Liu, B. Li, Y. Duan, H. Yang, and J. Hu. Exploring topic models in software engineering data analysis: A survey. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 357–362. IEEE, 2016.

[174] A. M. Sutii, M. van den Brand, and T. Verhoeff. Exploration of modularity and reusability of domain-specific languages: an expression DSL in metamod. *Computer Languages, Systems & Structures*, 2017.

[175] R. Tairas and J. Cabot. Cloning in DSLs: experiments with OCL. In *International Conference on Software Language Engineering*, pages 60–76. Springer, 2011.

[176] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[177] The Mathworks MATLAB/Simulink. `http://www.mathworks.com/products/simulink/`.

[178] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *31st Int. Conf. on Software Engineering*, pages 254–264. IEEE, 2009.

[179] E. Tromp and M. Pechenizkiy. Graph-based n-gram language identification on short texts. In *Proc. of the 20th Machine Learning conference of Belgium and The Netherlands*, pages 27–34, 2011.

[180] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11): 896–909, Nov 2006. ISSN 0098-5589. doi:10.1109/TSE.2006.112.

[181] S. Uchitel and M. Chechik. Merging Partial Behavioural Models. In *Int. Symposium on Foundations of Software Engineering*, pages 43–52. ACM, 2004. ISBN 1-58113-855-5.

[182] M. S. Uddin, C. K. Roy, and K. A. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 236–238. IEEE, 2013.

[183] M. van den Brand. Model Driven Software Engineering creates tomorrow's legacy, 2018. URL `http://gemoc.org/pub/20181015-GEMOC18/keynote-abstract.pdf`. Keynote at the 6th Int. Workshop on The Globalization of Modeling Languages co-located with MODELS 2018, Copenhagen, Denmark.

[184] B. van der Sanden, M. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, and R. Schiffelers. Modular model-based supervisory controller design for wafer logistics in lithography machines. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 416–425, Sept 2015. doi:10.1109/MODELS.2015.7338273.

[185] L. van der Sanden. *Performance analysis and optimization of supervisory controllers*. PhD thesis, Department of Electrical Engineering, 11 2018. Proefschrift.

[186] S. Wenzel and U. Kelter. Model-Driven Design Pattern Detection Using Difference Calculation. In *Proc. of the 1st Int. Workshop on Pattern Detection for Reverse Engineering*. IEEE, 2006. ISBN 0-7695-2719-1.

[187] R. Wester and J. Koster. The software behind moore's law. *IEEE Software*, 32(2):37–40, Mar 2015. ISSN 0740-7459.

[188] N. Weston, R. Chitchyan, and A. Rashid. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In *Proc. of the 13th Int. Software Product Line Conference*, pages 211–220. ACM, 2009.

[189] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Int. Conf. on Foundations of Software Engineering*, pages 314–323. ACM, 2000.

[190] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[191] F. Wild. *lsa: Latent Semantic Analysis*, 2015. URL `http://CRAN.R-project.org/package=lsa`. R package version 0.73.1.

[192] D. Wille. Managing Lots of Models: The FaMine Approach. In *Proc. of the 22nd ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 817–819. ACM, 2014. ISBN 978-1-4503-3056-5.

[193] D. Wille, S. Holthusen, S. Schulze, and I. Schaefer. Interface Variability in Family Model Mining. In *Proc. of the 17th Int. Software Product Line Conference Co-located Workshops*, pages 44–51. ACM, 2013. ISBN 978-1-4503-2325-3.

[194] D. Wille, S. Schulze, and I. Schaefer. Variability Mining of State Charts. In *Proc. of the 7th Int. Workshop on Feature-Oriented Software Development*, pages 63–73. ACM, 2016.

[195] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. Custom-Tailored Variability Mining for Block-Based Languages. In *IEEE 23rd Int. Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 271–282. IEEE, 2016.

[196] D. Wille, M. Tiede, S. Schulze, C. Seidl, and I. Schaefer. Identifying Variability in Object-Oriented Code Using Model-Based Code Mining. In *Int. Symposium on Leveraging Applications of Formal Methods*, volume 9953 of *LNCS*, pages 547–562. Springer, 2016. ISBN 978-3-319-47169-3.

[197] D. Wille, T. Runge, C. Seidl, and S. Schulze. Extractive Software Product Line Engineering Using Model-based Delta Module Generation. In *Proc. of the 11th Int. Workshop on Variability Modelling of Software-intensive Systems*, pages 36–43. ACM, 2017. ISBN 978-1-4503-4811-9.

[198] D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, and I. Schaefer. Improving custom-tailored variability mining using outlier and cluster detection. *Science of Computer Programming*, 163:62 – 84, 2018.

[199] Z. Wu and M. Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.

[200] Z. Xing. Model comparison with GenericDiff. In *Proc. of the IEEE/ACM Int. Cont. on Automated Software Engineering*, pages 135–138. ACM, 2010.

[201] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Int. Conf. on Automated Software Engineering*, pages 54–65. ACM, 2005. ISBN 1-58113-993-4.

[202] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[203] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, Dec. 1989. ISSN 0097-5397.

[204] X. Zhang, Ø. Haugen, and B. Møller-Pedersen. Model Comparison to Synthesize a Model-Driven Software Product Line. In *Int. Conf. on Software Product Line*, pages 90–99. IEEE, 2011. ISBN 978-1-4577-1029-2.

[205] P. Zikopoulos, C. Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[206] I. Žliobaitė, M. Pechenizkiy, and J. Gama. An overview of concept drift applications. In *Big data analysis: new algorithms for a new society*, pages 91–114. Springer, 2016.

Summary

Model Analytics and Management

Nowadays, we are witnessing an ever increasing complexity of software and systems, ranging from cyber-physical systems to the Internet of Things. Model-Driven Engineering (MDE) is a methodology which promotes the use of models, metamodels and model transformations as first-class citizens to tackle the complexity of building and maintaining those systems. With increased adoption of Model-Driven Engineering (MDE), however, the number of related artifacts in use, notably models, greatly increases. To confirm this, we present in this thesis quantitative evidence from both academia —in terms of repositories and datasets— and industry—in terms of large domain-specific language ecosystems. To be able to tackle this dimension of scalability in MDE, we propose to treat the artifacts as data, and apply various techniques—ranging from information retrieval to machine learning—to analyze and manage those artifacts in a holistic, scalable and efficient way.

Based on this idea, we have developed a framework called SAMOS (Statistical Analysis of MOdelS) to analyze, compare and visualize large datasets of models. The essence of the approach involves (1) extracting pieces of information (i.e. features) from models such as names, types and structure (e.g. linear chunks of n-grams, or subtrees); (2) defining comparison schemes (e.g. name comparison using Natural Language Processing for typos and synonyms, and structural comparison using edit distance) to obtain a Vector Space Model; and (3) applying distance measures and clustering to discover e.g. groups of similar models or model fragments. Working on different types of models such as EMF metamodels, state charts, feature models and industrial domain-specific models, we have used and evaluated SAMOS in various settings and application areas throughout the thesis.

Chapter 2 introduces the basics of our approach used in this thesis. We make a first step towards the handling of large datasets of models, EMF metamodels in particular, from a statistical perspective. Using VSM-based clustering of models represented as simple features including name and type information only, many characteristics and relations among the metamodels, such as clusters, sub-clusters and outliers, can be analyzed and visualized. We have explored two scenarios, namely model searching and repository exploration, for which we can utilize our approach. Particularly for the first case study, it is clearly noticeable that there are distinct outliers and groupings in the search results. This information can be used to improve the navigation or precision of the search results. The second case study, on the other hand, deals with a heterogeneous set of domains and allows identifying domains, subdomains and also the proximities between related ones. This grouping information can be used for domain model recovery as well as model repository management scenarios.

We have extended SAMOS to incorporate structural context into clustering in Chapter 3. We have indicated a shortcoming of the basic approach as in the previous chapter, i.e. ignoring the context of model elements, and have proposed an n-gram based representation and comparison, which can be considered as the compromise between context-less clustering approaches and advanced pairwise structural techniques. We have evaluated our approach on an Ecore dataset. We have shown that n-grams improve the clustering accuracy on average. Picking an $n > 1$ is shown to increase complexity (though not monotonically) and using $n = 2$ is suggested for smaller datasets and precision-oriented tasks.

In Chapter 4, we have introduced how our approach can be used to improve the results of variability mining from models of block-based languages. For this purpose, we have demonstrated and discussed how our cluster and outlier detection can improve the variability information generated by the family mining approach developed by our collaboration partners. Using the presented extension it is possible to remove outliers (e.g., completely unrelated variants) from a set of input models, i.e. state charts, and cluster them into more meaningful sets (e.g., relevant for particular users).

We have presented in Chapter 5 an application of our generic model clustering technique to comparing feature models. With two exploratory case studies on the 1034-model dataset in the S.P.L.O.T. repository, we get (1) a repository overview and major domains therein, (2) very similar models in the repository such as duplicates and clones. Based on the studies, we conclude that our approach can help with the use and maintenance of emerging repositories such as S.P.L.O.T. The clone detection part is properly treated later in Chapter 6. There we have extended SAMOS with additional scoping, feature extraction and comparison schemes, customized distance measures and clustering algorithms in the context of metamodel clone detection. We have evaluated our approach using a variety of case studies involving both synthetic and real data; and identified the strengths and weaknesses of our approach along with two other state-of-the-art clone detectors. We conclude that SAMOS stands out with its higher accuracy while still being substantially scalable.

In Chapter 7, we have applied our approach in an industrial context, with various analyses on ASML's MDE ecosystems. We have used and extended SAMOS to operate on ASML's languages and models. We have elaborated the domain-specific extension of SAMOS, specifically for ASML's ASOME data and control models to enable clone detection on those models. In extensive case studies, we have performed clone detection on ASML's models, and additionally language-level analyses ranging from cross-DSL conceptual analysis and clone detection to architectural analysis for the CARM2G ecosystem. We have presented our findings along with valuable feedback from the domain experts on the nature of cloning in the ecosystems, and indicated opportunities such as refactoring to support the maintenance and quality of the ever-growing and evolving ecosystems.

Moreover, we have experimented with a distributed data processing back-end for SAMOS (Chapter 8), as a means to improve its scalability further, to help with e.g. larger datasets or more expensive analyses. Along with integration into the Eclipse KNIME data mining ecosystem, this is part of our efforts in transforming SAMOS into a mature open framework to be used and extended in further scenarios.

Curriculum Vitae

Personal Information

Name: Önder Babur

Date of birth: February 5, 1985

Place of birth: Ankara, Turkey

Education

2014–now *Ph.D candidate,* Eindhoven University of Technology, The Netherlands

2007–2010 *M.Sc in Software Systems Engineering,* RWTH Aachen, Germany

2002–2007 *B.Sc in Computer Engineering,* Middle East Technical University, Turkey

Work Experience

2017–now *Part-time visiting researcher,* ASML, The Netherlands

2012–2013 *Research intern,* IMDEA Software Institute, Spain

2011–2012 *Software systems engineer,* IVU Traffic Technologies AG, Germany

Organization and Community Service

4TU.NIRICT community building project on Model Management and Analytics, acquired 23.4k Euros funding (2017), founded `http://modelanalytics.wordpress.com`

Editor, Book on Model Management and Analytics for Large Scale Systems, Elsevier (2019)

Chair, Int. Workshop on Analytics and Mining of Model Repositories @ MODELS'18

Chair, Special Session on Model Management and Analytics @ MODELSWARD'18

Chair, Dutch Symposium on Model Management and Analytics, WUR (2017)

Proceedings chair MODELS'18, publicity chair VaMoS'17, student volunteer ETAPS'16

Titles in the IPA Dissertation Series since 2015

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03