Bridging the Data Desert: Mitigating Challenges of Model Accessibility in Simulink Research

Inaugural dissertation

of the Faculty of Science,

University of Bern

presented by

Alexander Boll

Supervisor of the doctoral thesis:

Prof. Dr. Timo Kehrer

University of Bern

Bridging the Data Desert: Mitigating Challenges of Model Accessibility in Simulink Research

Inaugural dissertation

of the Faculty of Science,

University of Bern

presented by

Alexander Boll

Supervisor of the doctoral thesis:

Prof. Dr. Timo Kehrer

University of Bern

Accepted by the Faculty of Science.

Bern, December 15, 2025

The Dean

Prof. Dr. Jean-Louis Reymond

Supervisor and First Reviewer
Prof. Dr. Timo Kehrer
University of Bern

Second Reviewer
Prof. Dr. Christoph Csallner
University of Texas at Arlington

Abstract

Simulink is the industry standard for model-driven development in safety-critical domains such as automotive, aerospace, and medical devices. However, empirical research in the context of Simulink faces a persistent challenge: a scarcity of high-quality, industry-representative models that are essential for rigorous tool evaluation, empirical validation, and reproducible studies. This scarcity not only slows down scientific progress but also contributes to a reproduction crisis in the field – primarily due to the unavailability of experimental models.

This thesis addresses this challenge through three interconnected contributions, grounded in a multi-method approach that includes a systematic literature review, empirical case studies, community surveys, dataset analysis, and tool prototyping and validation:

1. A diagnosis of model scarcity demonstrating that the lack of models limits the ability to conduct empirical research and also contributes to only 9% of Simulink tool studies meeting reproducibility criteria (*i.e.*, all artifacts available).

2. An assessment of existing open-source Simulink models and datasets, evaluating their suitability for empirical research and investigating their limitations in scale, complexity, and industrial realism. Through case studies – including model matching, analyzing bus architecture of Simulink models, and investigating commenting practices – we demonstrate that open-source models, while imperfect, can serve as valuable research subjects for empirical investigation when carefully selected and used appropriately.

3. To address the lack of (i) large-scale and (ii) industry-representative models, we developed two novel tools: (i) GRANDSLAM, a linearly scaling synthesizer for Simulink that generates models with adjustable properties, enabling the synthesis of very large open-source models; (ii) SMOKE, a model anonymizer that removes sensitive information from Simulink models while preserving their structural properties, thus facilitating the sharing of real-world models without violating intellectual property constraints.

Our work complements and extends contemporary datasets by showing their suitability for empirical research and providing tools for their expansion. By lowering the barriers to data access, we advance open science in model-driven engineering, enabling reproducible studies, specifically large-scale studies that were previously infeasible. The contributions of this thesis are foundational: they narrow the "data desert" in Simulink research and foster collaboration through shareable resources. Beyond immediate applications, our tools and findings support standardized benchmarks, comparative tool evaluations, and longitudinal studies of modeling practices – ultimately strengthening the empirical rigor and industrial relevance of Simulink research.

In summary, this thesis provides both the evidence of a critical gap in Simulink research and practical solutions to address it, offering a pathway toward more transparent, reproducible, and impactful model-driven engineering.

Acknowledgments

I am profoundly grateful for the incredible people and opportunities that have shaped my PhD journey. This thesis would not have been possible without their support, collaboration, and encouragement.

First and foremost, I owe my deepest gratitude to my advisor, Prof. Dr. Timo Kehrer. His ability to find solutions for every challenge, combined with his unwavering engagement and liberal supervision style created the perfect environment for me to thrive. Timo's guidance and trust allowed me to explore my ideas freely, and I will always cherish the time I spent under his wings. Collaborating with him has been a privilege and a constant source of inspiration. Any introduction without his revision feels unfinished by default.

I am also sincerely thankful to my second reviewer, Prof. Dr. Christoph Csallner, for his willingness to step into this role and for the enriching collaborations we have shared. His spontaneous agreement to support this process is greatly appreciated.

My colleagues have been instrumental in making this journey both productive and enjoyable. To Dr. Alexander Schultheiß, whose brilliant mind generated countless ideas that enriched our work, and to Roman Bögli, whose collaborations are as successful as they are enjoyable – thank you for the great times, both in and out of the office. Roman, your hospitality during my stay in your apartment and your "school runs" to the university will not be forgotten. A special mention goes to my fellow open science advocate, Dr. Sohil Shrestha, with whom I shared not only impactful collaborations but also a commitment to advancing our field.

This thesis and my other work have benefited immensely from the input of many collaborators. I extend my heartfelt thanks to Florian Brokhausen, Dr. Tiago Amorim, Prof. Dr. Andreas Vogelsang, Nicole Vieregg, Dr. Paul Maximilian Bittner, Prof. Dr. Lars Grunske, Prof. Dr. Thomas Thüm, Prof. Dr. Pooja Rani, Dr. Manuel Ohrndorf, Dr. Shafiul Azam Chowdhury, Prof. Dr. Michael Goedicke, Yael van Dok, Pablo Valenzuela-Toledo, Chuyue Wu, Sandro Hernández, Roman Macháček, Sebastiano Panichella, Sandra Greiner, Gregorio Dalia, Andrea di Sorbo, Corrado Aaron Visaggio, and Dr. Kim Völlinger. Your insights and contributions have elevated our work in ways I could not have achieved alone.

To my family, thank you for the upbringing and support that gave me the privilege to pursue academia. The sibling peer pressure certainly had a positive influence in propelling me into the doctorate club.

To my girlfriend, whose faith in me often surpassed my own: thank you for standing by me through every high and low. Your love and encouragement have been my anchor.

Finally, I am grateful to the German and Swiss citizens whose taxes funded my PhD. Your investment in education and research has made this journey possible.

Contents

List of Publications

During his doctoral studies, the author contributed to several peer-reviewed publications. Below is a list of articles and papers resulting from his doctoral studies, including manuscripts currently under review.

📘 Journal Article 📄 Conference Paper 🔧 Tool Demonstration Paper
⌛ Under Review * Co-first Authorships

Main Works

This thesis consists of the following works, given in chronological order:

[1] **Alexander Boll**, Florian Brokhausen, Tiago Amorim, Timo Kehrer, and Andreas Vogelsang. "Characteristics, potentials, and limitations of open-source Simulink projects for empirical research". In: *Software and Systems Modeling*. 20. 6. (Apr. 2021). Pp. 2111–2130. DOI: 10.1007/s10270-021-00883-0. 📘

[2] **Alexander Boll**, Nicole Vieregg, and Timo Kehrer. "Replicability of experimental tool evaluations in model-based software and systems engineering with MATLAB/Simulink". In: *Innovations in Systems and Software Engineering*. 20. 3. (Mar. 2022). Pp. 209–224. DOI: 10.1007/s11334-022-00442-w. 📘

[3] Alexander Schultheiß, Paul Maximilian Bittner, **Alexander Boll**, Lars Grunske, Thomas Thüm, and Timo Kehrer. "RaQuN: a generic and scalable n-way model matching algorithm". In: *Software and Systems Modeling*. 22. 5. (Nov. 2022). Pp. 1495–1517. DOI: 10.1007/s10270-022-01062-5. 📘

[4] Tiago Amorim*, **Alexander Boll***, Ferry Bachman, Timo Kehrer, Andreas Vogelsang, and Hartmut Pohlheim. "Simulink bus usage in practice: an empirical study". In: *The Journal of Object Technology*. 22. 2. (July 2023). 2:1–14. DOI: 10.5381/jot.2023.22.2.a12. The 19th European Conference on Modelling Foundations and Applications (ECMFA 2023). 📘

[5] Sohil Lal Shrestha, **Alexander Boll**, Timo Kehrer, and Christoph Csallner. "ScoutSL: An Open-Source Simulink Search Engine". In: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, Oct. 2023. Pp. 70–74. DOI: 10.1109/MODELS-C59198.2023.00022. 🔧

[6] **Alexander Boll**, Pooja Rani, Alexander Schultheiß, and Timo Kehrer. "Beyond code: Is there a difference between comments in visual and textual languages?" In: *Journal of Systems and Software*. 215. (Sept. 2024). Pp. 1–17. DOI: 10.1016/j.jss.2024.112087. 📘

[7] **Alexander Boll**. "GRANDSLAM: Linearly Scalable Model Synthesis". In: *19th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2026.* (under review)

[8] **Alexander Boll**, Manuel Ohrndorf, and Timo Kehrer. "SMOKE2.0 Whitebox Anonymizing Intellectual Property in Models While Preserving Structure". In: *Journal of Software and Systems Modeling*. (under revision).

Related Works

The following publications relate to this thesis but are not part of the main body. The conference papers ([9] and [11]) are fully incorporated and extended in their respective journal publications ([2] and [8]), which provide a more comprehensive and updated treatment. Additionally, while [10] represents valuable research, the author's contributions to this work do not directly align with the focus of this thesis.

[9] **Alexander Boll** and Timo Kehrer. "On the Replicability of Experimental Tool Evaluations in Model-Based Development". In: *Systems Modelling and Management*. Cham: Springer International Publishing, 2020. Ed. by Önder Babur, Joachim Denil, and Birgit Vogel-Heuser. Pp. 111–130.

[10] Sohil Lal Shrestha, **Alexander Boll**, Shafiul Azam Chowdhury, Timo Kehrer, and Christoph Csallner. "EvoSL: A Large Open-Source Corpus of Changes in Simulink Models & Projects". In: *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, Oct. 2023. Pp. 273–284. DOI: 10.1109/MODELS58315.2023.00024.

[11] **Alexander Boll**, Timo Kehrer, and Michael Goedicke. "SMOKE: Simulink Model Obfuscator Keeping Structure". In: *MODELS Companion '24: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. Linz, Austria: ACM, Oct. 2024. Pp. 41–45. DOI: 10.1145/3652620.3687788.

Other Works

In addition, the author has contributed to several works outside the scope of this thesis, which advance research in other areas of software engineering.

[12] Alexander Schultheiß, **Alexander Boll**, and Timo Kehrer. "Comparison of Graph-based Model Transformation Rules". In: *Journal of Object Technology*. 19. 2. (July 2020). 3:1–21. DOI: 10.5381/jot.2020.19.2.a3. The 16th European Conference on Modelling Foundations and Applications (ECMFA 2020).

[13] **Alexander Boll***, Yael Van Dok*, Manuel Ohrndorf*, Alexander Schultheiß, and Timo Kehrer. "Towards Semi-Automated Merge Conflict Resolution: Is It Easier Than We Expected?" In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. Salerno, Italy: ACM, June 2024. Pp. 282–292. DOI: 10.1145/3661167.3661197.

[14] Roman Bögli, **Alexander Boll**, Alexander Schultheiß, and Timo Kehrer. “Beyond Software Families: Community-Driven Variability”. In: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. New York, NY, USA: ACM, July 2025. Pp. 571–575. DOI: 10.1145/3696630.3728501.

[15] Pablo Valenzuela-Toledo, Chuyue Wu, Sandro Hernández, **Alexander Boll**, Roman Machacek, Sebastiano Panichella, and Timo Kehrer. “Explaining GitHub Actions Failures with Large Language Models: Challenges, Insights, and Limitations”. In: *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE, Apr. 2025. Pp. 286–297. DOI: 10.1109/ICPC66645.2025.00037.

[16] Alexander Schultheiß, **Alexander Boll**, Paul Maximilian Bittner, Sandra Greiner, Thomas Thüm, and Timo Kehrer. “Decades of GNU Patch and Git Cherry-Pick: Can We Do Better?” In: *Proceedings International Conference on Software Engineering (ICSE)*. 2026. (accepted)

[17] Manuel Ohrndorf*, **Alexander Boll***, Roman Bögli, and Timo Kehrer. “Turning Merge Conflicts Into Conflict-Induced Variability”. In: *Proceedings International conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. (accepted)

[18] Pablo Valenzuela-Toledoa, Gregorio Dalia, **Alexander Boll**, Andrea Di Sorbo, Corrado Aaron Visaggio, Timo Kehrer, and Sebastiano Panichella. “Vulnerability-Proneness of GitHub Actions: A Pre-Adoption Approach for Security Risk Assessment”. In: *Journal of Systems and Software, Special Issue: Intelligent DevOps for Performance, Sustainability, and Reliability*. (under review)

[19] Roman Bögli, **Alexander Boll**, Alexander Schultheiß, and Timo Kehrer. “Community-Driven Variability: Characterizing a new Software Variability Paradigm”. In: *Automated Software Engineering*. (in revision).

Author Contributions

Thesis-related List of Author Contributions

Table 1 outlines the author's estimated contributions to the thesis-related works, using the CRediT taxonomy [335] as a framework. An additional "Dissemination" role is included to account for conference presentations. The roles "Resources", "Funding acquisition", and "Supervision" are omitted, as they do not apply to the author's contributions. The estimates are approximate and subjective, reflecting the author's assessment of their relative contribution. These estimates provide a transparent yet simplified overview of collaborative work, as precise quantification was often impractical. Pie segments indicate contribution levels as follows: ○ no contribution, ◔ partial contribution or equal contribution among three or more contributors, ◑ significant contribution or equal contribution between two contributors, ◕ major contribution, ● sole contribution, and — not applicable.

Table 1. Author contribution for main and thesis-related works.

Work in thesis section	2.1	2.2	3.1	3.2	3.3	3.4	4.1	4.2	—
Reference	[9, 2]	[5]	[1]	[3]	[4]	[6]	[7]	[11, 8]	[10]
Conceptualization	◑	◔	◔	○	◔	◑	●	●	○
Data curation	◕	◔	◕	◔	◑	◑	●	●	○
Formal analysis	●	◔	◕	○	◑	◕	●	●	○
Investigation	◕	◔	◑	◔	◑	◑	●	●	◔
Methodology	◑	◔	◔	○	◔	◑	●	●	○
Project administration	◑	○	◔	○	◔	◑	●	●	○
Software	●	○	◕	◔	●	●	●	●	○
Validation	●	◔	◑	○	◑	◕	●	●	◔
Visualization	●	◑	◕	○	◕	●	●	●	○
Writing – original draft	◑	◔	◔	○	◑	◕	●	◕	○
Writing – review & editing	◑	◔	◔	◔	◔	◔	◕	◑	◔
Dissemination	●	●	—	○	○	—	—	●	●

Legend

2.1: Replicability of experimental tool evaluations in model-based software and systems engineering with MATLAB/Simulink, On the Replicability of Experimental Tool Evaluations in Model-Based Development
2.2: ScoutSL: An Open-Source Simulink Search Engine
3.1: Characteristics, potentials, and limitations of open-source Simulink projects for empirical research
3.2: RaQuN: a generic and scalable n-way model matching algorithm
3.3: Simulink bus usage in practice: an empirical study
3.4: Beyond code: Is there a difference between comments in visual and textual languages?
4.1: GRANDSLAM: Linearly Scalable Model Synthesis
4.2: SMOKE2.0 Whitebox Anonymizing Intellectual Property in Models While Preserving Structure, SMOKE: Simulink Model Obfuscator Keeping Structure
—: EvoSL: A Large Open-Source Corpus of Changes in Simulink Models & Projects

List of Academic Contributions

Beyond the publications listed in the list of publications, the author has contributed to the software engineering community through peer review, service roles, and student supervision.

Peer Review Activities

Table 2 provides a chronological summary of the author's subreviewer roles at various academic journals and conferences.

Table 2. Author subreviewer activities by year and venue.

Year	Venue	# Papers
2020	ASE	2
	JOT	1
	SPE	1
2021	ASE	1
	J.UCS	1
2022	ICSE (NIER track)	3
2023	MODELS	1
	ESWA	1
	SPE	1
2024	TSE	1
	ICSE	1
	VaMoS	1
2025	EASE	1
	ICSME	1
	ICSE	2
	FSE	2

Legend

ASE – International Conference on Automated Software Engineering

JOT – Journal of Object Technology

SPE – Software: Practice and Experience

J.UCS – Journal of Universal Computer Science

ICSE – International Conference on Software Engineering

MODELS – International Conference on Model Driven Engineering Languages and Systems

ESWA – Expert Systems With Applications

TSE – Transactions on Software Engineering

VaMoS – International Working Conference on Variability Modelling of Software-Intensive Systems

EASE – International Conference on Evaluation and Assessment in Software Engineering

ICSME – International Conference on Software Maintenance and Evolution

FSE – International Conference on the Foundations of Software Engineering

Service and Supervision

The author also contributed through:

- **Conference roles:** Web chair for *VaMoS 2024* and artifact reviewer for *VaMoS 2025*.
- **Student supervision:** Main supervisor for two BSc theses and one MSc thesis.

1 Introduction

Domain-specific models are central to model-driven development for complex, software-intensive systems [139, 211]. These models not only act as core abstractions in early-stage analysis, simulation, and validation, but also serve as the foundation for automated code generation. Matlab/Simulink[1] (hereafter referred to as Simulink) has become the industry-standard platform for model-driven embedded system development across fields such as automotive, aerospace, industrial automation, and medical devices [83, 208].

In light of Simulink's importance in such safety-critical fields, the research community is active in developing innovative methods for managing and processing models in order to ease or enable various aspects of Simulink development [64, 116, 126, 177, 207, 230, 250, 251, 272, 273, 362]. These methods are typically implemented in a tool. While the theoretical and conceptual advancements of the methods are essential, the tools' practical value also relies on empirical validation, typically done through experiments [38, 49]. To conduct such validations, researchers critically need suitable experimental data – in our case Simulink models. These models are essential not only for experiments but also for sharing results in compliance with FAIR principles (Findable, Accessible, Interoperable, Reusable) [201, 268], thereby fostering transparency, verification, and reproduction [41, 90, 369].

However, the Simulink research community faces a persistent challenge: a scarcity of open-source models, particularly those considered "real world", or "industry-like"[2] models [208, 218, 234, 267]. Simulink models developed and maintained in an industrial context are often withheld from public access due to their sensitive intellectual property or license restrictions [124, 146, 228, 283]. Many studies have deemed existing open-source models insufficient in realism or scale [218, 267, 283], unlike classical textual or code-based development, where large, well-curated open-source collections are readily available [169, 185]. Although prior work has acknowledged the model deficit, its impact on Simulink research had not been systematically studied. To address this gap, the first goal of this thesis (Goal $\mathcal{A}$) is to systematically characterize the extent of model scarcity and quantify its impact on Simulink research.

[1] `https://www.mathworks.com/products/simulink` (visited on 15/12/2025)

[2] While no formal definition for "industry-like" exists, we observe that researchers typically associate the term with models of a certain scale, complexity, and applicability in industrial settings.

A straightforward approach to addressing model scarcity is to curate collections of open-source models and make them publicly available [236, 337, 352, 10, 363]. Such collections – or datasets – offer multiple benefits, including: (i) experimental use, such as evaluating novel tools, enabling third-party verification and reproduction, or comparing methods/tools on established datasets; (ii) empirical research use, such as observing and demonstrating findings on open-source datasets or studying model development processes and evolution; and (iii) preservation and sampling, such as conserving valuable models (*e.g.*, from otherwise deleted GitHub repositories) or estimating population sizes for sampling methods [232].

However, simply amassing open-source models does not guarantee high-quality datasets suitable for all applications. While many such models may not reflect best practices, they still reflect common practices – studying which can yield valuable insights. Without careful curation, though, these datasets may remain unsuitable for tasks requiring high-quality or representative models. Baltes and Ralph [289] recommend that a curated corpus – in our case, a model dataset – should include models that are: (i) diverse in domain (*e.g.*, aerospace, automotive, robotics); (ii) versatile in use (*e.g.*, for design, simulation, or code generation); (iii) sufficiently large for sampling methods (*e.g.*, heterogeneity sampling, bootstrapping); and (iv) aligned with target parameters (*e.g.*, size, quality, or time under development). Yet, it remained unclear whether existing model datasets met these criteria. To address this, the second goal of this thesis (Goal $\mathcal{B}$) is to assess open-source Simulink models and datasets and evaluate their suitability for empirical research.

The third goal of this thesis ($\mathcal{C}$) goes beyond $\mathcal{A}$ and $\mathcal{B}$ by tackling the scarcity of large-scale, industry-representative Simulink models. Instead of merely acknowledging this gap, we develop tools and methods to acquire and generate models that meet the demanding criteria for empirical research – thereby enabling studies previously constrained by artifact availability.

Our three goals structure the three main chapters of this thesis, as illustrated in Figure 1.1.

Chapter 2 (Goal $\mathcal{A}$) We diagnosed and verified the aforementioned model scarcity by: (i) Conducting a systematic reproduction study (see Section 2.1) showing that only 9% of Simulink tool studies were reproducible – having all artifacts available – mostly because they did not publish the models from their evaluation. One major contributing factor for this was the lack of suitable open-source models for the study in the first place. (ii) We conducted a survey (see Section 2.2) within the Simulink research community where the majority of respondents confirmed having problems in acquiring suitable Simulink models for their research. Our findings in Chapter 2 are the key motivation for the rest of this thesis' works.

Chapter 3 (Goal $\mathcal{B}$) We first investigated the (at the time) largest dataset of Simulink models [236] with the focus on whether the models are suitable for empirical research (see Section 3.1). We found that they are diverse, and often suitable for research purposes but also noted a lack of large and industry-like models. We conclude that the datasets are generally suitable, provided that researchers carefully sample from them according to their research needs. We further demonstrated their practical suitability through three case studies: evaluating a model matching tool (see Section 3.2), analyzing commenting practices, and studying bus element usage (see Sections 3.3 and 3.4).

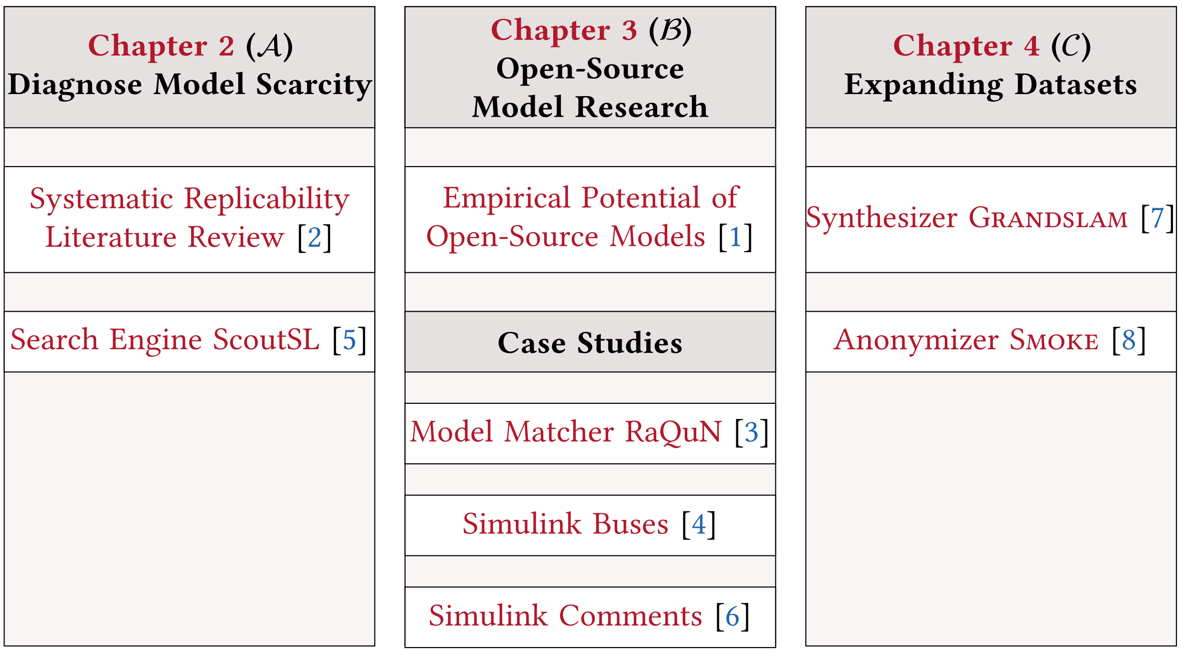

Chapter 2 ($\mathcal{A}$)
Diagnose Model Scarcity
Systematic Replicability Literature Review [2]
Search Engine ScoutSL [5]
Chapter 3 ($\mathcal{B}$)
Open-Source Model Research
Empirical Potential of Open-Source Models [1]
Case Studies
Model Matcher RaQuN [3]
Simulink Buses [4]
Simulink Comments [6]
Chapter 4 ($\mathcal{C}$)
Expanding Datasets
Synthesizer GRANDSLAM [7]
Anonymizer SMOKE [8]

Figure 1.1. A roadmap of the contributions covered in this thesis. In abstract terms, the chapters represent a diagnosis of challenges in our domain (Chapter 2), theoretical insights and practical demonstrations of what is possible despite the challenges (Chapter 3), and mitigation methods for the challenges (Chapter 4).

Chapter 4 (Goal $\mathcal{C}$) We presented two novel mitigations addressing the gaps in currently existing Simulink model datasets. (1) GRANDSLAM, the first model synthesizer that scales linearly, enabling the extension of Simulink model datasets with large-scale models measuring more than one million elements (see Section 4.1). (2) SMOKE, a model anonymization tool, which removes sensitive information from industrial Simulink models, while preserving the structural properties. It facilitates the sharing of real-world models – or at least research-valuable parts of them – without violating intellectual property constraints. SMOKE is intended to foster collaboration between industry and researchers, enabling research on realistic models (see Section 4.2).

While the primary beneficiaries of this thesis results are researchers in the model-driven engineering field, its contributions extend beyond academia. The existing open-source Simulink models – whose wide potential we demonstrate – can serve as pedagogical resources, enabling students and practitioners to learn modeling practices from diverse, practical examples. GRANDSLAM, our scalable model synthesizer, has already demonstrated its utility in fuzzing and scalability testing – uncovering 11 confirmed bugs and 2 scaling issues that MathWorks intends to address, thus directly improving the robustness of industrial tools. Finally, SMOKE, our anonymization tool, paves the way for future industry-research collaborations by enabling the secure sharing of proprietary models, bridging the gap between academic research and industrial practice. Together, these contributions not only advance empirical research but also foster broader adoption and collaboration across domains.

With our thesis contributions, we assert the key challenge of acquiring suitable Simulink models and its impact (Goal $\mathcal{A}$), investigate and demonstrate how empirical research is possible despite this challenge (Goal $\mathcal{B}$), and develop two innovative approaches to mitigating the challenge (Goal $\mathcal{C}$). Overall, we foster research by showing the value of existing open-source models as either empirical study subjects or tool evaluation subjects. We further provide researchers with two tools for either synthesizing large models, or deriving them from actual industrial models. Our work complements contemporary advancements in the field, particularly the expansion and refinement of model datasets [337, 352, 10]. Together, these advancements help to bridge the "data desert" in our field and are foundational to open science [297].

2

Diagnosing Simulink Model Scarcity

This chapter lays the motivational foundation for the following parts of the thesis and consists of two publications: *Replicability of experimental tool evaluations in model-based software and systems engineering with MATLAB/Simulink* (Section 2.1) and *ScoutSL: An Open-Source Simulink Search Engine* (Section 2.2).

Experiencing Model Scarcity Our initial motivation in investigating the Simulink model scarcity and reproduction problems in the Simulink field was our associated industry partner in our first Simulink related research project that declined their collaboration. They took a key ingredient for our research's feasibility with them: their Simulink models. These industrial models are valuable and contain intellectual property the associated partner was no longer willing to share in order to protect it. For our research project, we were thus left without experimental subjects. We could neither gather empirical insights from the models themselves, nor verify any of our research prototypes we built in realistic settings.

Investigating Community Practices for Model Acquisition We thus (at first unsystematically) set out to learn how other teams in the research community acquired their models. If we can access their models, we may use them for our own experiments. Alternatively, we may try to emulate methods of acquiring models that worked for other teams. However, we quickly found many indications that other researchers also struggled to obtain their experimental subjects. The authors noted challenges in conducting their studies, *e.g.*, "Crucial to our main study, we planned to work on real industrial data (this is an obstacle for most studies due to proprietary intellectual property concerns)." [208].

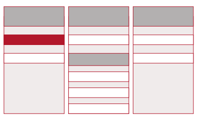
Systematic Literature Review To better understand the scope and impact of this issue, we thus conducted the systematic literature review *Replicability of experimental tool evaluations in model-based software and systems engineering with MATLAB/Simulink.*[1] In this work, we

[1]Note, that since we published this work, the terms 'replicability' and 'reproducibility' got switched up by the Association for Computing Machinery. We now use the most up to date terminology in the text, apart from the publication titles themselves. Thus 'reproducibility' for in this dissertation denotes another research team using the same data to achieve the same results. 'Replicability' means another research team using different

systematically investigated from where researchers acquired their models (*e.g.*, 25% from research, 18% from industry partners), and some basic characteristics of the models (*e.g.*, only three models used for evaluation on average, most created for single-use demonstrations). We further collected and analyzed studies that report on methods, techniques, or algorithms for MATLAB/Simulink development. We studied whether their research is reproducible, at least *in principle*. For this criterion we required (i) their software artifacts to be made available, (ii) the experimental data (Simulink models) that were used to be made available. Overall, we found that only 9% of the studies shared both their software and research data at the same time. In most cases, the models were not accessible. In addition, even when authors held the rights and technical ability to share their own tools, these were frequently not made available to the public. Our follow-up analysis following the reproduction packages' directions revealed that *none* of the studies were reproducible in practice.

Survey on Contemporary Challenges, ScoutSL Literature reviews are inherently retrospective. One key aspect of our publication *ScoutSL: An Open-Source Simulink Search Engine* is our investigation of the contemporary challenges for researchers in acquiring suitable Simulink models for their studies. We conducted a survey with 15 researchers who previously published on Simulink tools and their evaluation. This survey was partly aimed at clarifying the needs of the research community for a Simulink model search engine. However, it further supports this thesis' motivation, *e.g.*, it showed that the majority of researchers often had difficulties finding suitable Simulink models or projects for their research and specified some of the properties models have to fulfill to be suitable. The main result of our study, ScoutSL, a Simulink model search engine, hosted at https://scoutsl.net, makes it easier to find and access suitable models from datasets such as the ones described in Chapter 3. ScoutSL improves the accessibility to open-source models – a key step toward mitigating the scarcity problem.

Summary and Context This chapter establishes the urgency of the model scarcity problem, motivating our solutions: assessing open-source models and their suitability (Section 3.1), improving accessibility (Section 2.2), and enabling synthesis/anonymization for dataset expansion (Sections 4.1 and 4.2).



↪ Thesis Roadmap

2.1 Reproducibility of Experimental Tool Evaluations in Model-Based Software and Systems Engineering with MATLAB/Simulink

Authors: Alexander Boll, Nicole Vieregg, and Timo Kehrer.
Published in: Innovations in Systems and Software Engineering, 2022.
Copyright: © 2022 The Authors.
Extends: *On the Replicability of Experimental Tool Evaluations in Model-Based Development*, Alexander Boll and Timo Kehrer, In: *Systems Modelling and Management*, 2020.

data to achieve the same research results. https://www.acm.org/publications/policies/artifact-review-and-badging-current. (visited on 15/12/2025)

Abstract: Research on novel tools for model-based development differs from a mere engineering task by not only developing a new tool, but by providing some form of evidence that it is effective. This is typically achieved by experimental evaluations. Following principles of good scientific practice, both the tool and the models used in the experiments should be made available along with a paper, aiming at the reproducibility of experimental results. We investigate to which degree recent research reporting on novel methods, techniques, or algorithms supporting model-based development with MATLAB/Simulink meets the requirements for reproducibility of experimental results. Our results from studying 65 research papers obtained through a systematic literature search are rather unsatisfactory. In a nutshell, we found that only 31% of the tools and 22% of the models used as experimental subjects are accessible. Given that both artifacts are needed for a reproduction study, only 9% of the tool evaluations presented in the examined papers can be classified to be reproducible in principle. We found none of the experimental results presented in these papers to be fully reproducible, and 6% partially reproducible. Given that tools are still being listed among the major obstacles of a more widespread adoption of model-based principles in practice, we see this as an alarming signal. While we are convinced that this situation can only be improved as a community effort, this paper is meant to serve as starting point for discussion, based on the lessons learned from our study

2.1.1 Introduction

Model-based development [139, 211] is a much appraised and promising methodology to tackle the complexity of modern software-intensive systems, notably for embedded systems in various domains such as transportation, telecommunications, or industrial automation [83]. It promotes the use of models in all stages of development as a central means for abstraction and starting point for automation, *e.g.*, for the sake of simulation, analysis or software production, with the ultimate goal of increasing productivity and quality.

Consequently, model-based development strongly depends on good tool support to fully realize its manifold promises [64]. Research on model-based development often reports on novel methods and techniques for model management and processing which are typically embodied in a tool. In addition to theoretical and conceptual foundations, some form of evidence is required concerning the effectiveness of these tools, which typically demands for an experimental evaluation [38, 49]. In turn, experimental results should be reproducible in order to increase the validity and reliability of the outcomes observed in an experiment [41, 90]. Therefore, both the tool and the experimental subject data, essentially the models used in the experiments, should be made available following the so-called FAIR principles—Findability, Accessibility, Interoperability, and Reusability [201, 268]—aiming at the reproducibility of experimental results.

In this paper, we investigate to which degree recent research on tools for model-based development of embedded systems meets the requirements for reproducibility of experimental results. We focus on tools for MATLAB/Simulink (referred to as Simulink, for short), which has emerged as a de-facto standard for automatic control and digital signal processing. In particular, we strive to answer the following research questions:

RQ1: Are the experimental results of evaluating tools supporting model-based development with Simulink reproducible?

RQ2: From where do researchers acquire Simulink models used as experimental subjects and what are their basic characteristics?

RQ3: Does the reproducibility of experimental results correlate to the impact of a paper?

We conduct a systematic literature review in order to compile a list of relevant papers from which we extract and synthesize the data to answer these research questions. Starting from an initial set of 942 papers that matched our search queries on the digital libraries of IEEE, ACM, ScienceDirect and dblp, we identified 65 papers which report on the development and evaluation of a tool supporting Simulink, and for which we did an in-depth investigation. Details of our research methodology, including the search process, paper selection and data extraction, are presented in Section 2.1.2.

In a nutshell, we found that models used as experimental subjects and prototypical implementations of the presented tools, both of which are essential for reproducing experimental results, are accessible for only a minor fraction (namely 22% and 31%) of the investigated papers. We further found the results for none of these papers to be fully reproducible (RQ1), achieving only partial reproducibility for 6%. The models come from a variety of sources, *e.g.*, from other research papers, industry partners of a paper's authors, open source projects, or examples provided along with Simulink or any of its toolboxes. Interestingly, the smallest fraction of models (only 3%) is obtained from open source projects, and the largest one (about 18%) is provided by industrial partners (RQ2). While we think that, in general, the usage of industrial models strengthens the validity of experimental results, such models are often not publicly available due to confidentiality agreements. These findings are confirmed by other research papers which we investigated during our study. Finally, we found papers having a better reproducibility also being cited more often (RQ3), confirming results of [69, 134]. Our results are presented in detail in Section 2.1.3.

While we do not claim our results to represent a complete image of how researchers adopt the FAIR principles of good scientific practice and deal with the reproducibility of experimental results in our field of interest (see Section 2.1.4 for a discussion of major threats to validity), we see our findings as an alarming signal. Given that tools are still being listed among the major obstacles of a more widespread adoption of model-based principles in practice [141], we need to overcome this "[reproducibility] problem" in order to make scientific progress. We are strongly convinced that this can only be achieved as a community effort. The discussion in Section 2.1.5 is meant to serve as a starting point for this, primarily based on the lessons learnt from our study. Finally, we review related work in Section 2.1.6, and Section 2.1.7 concludes the paper.

This paper is a revised version of our previous conference paper [9], providing the following extensions:

- In our previous work, we only investigated the principal reproducibility of the experimental results presented in the research papers compiled by our systematic literature search. In this extended version, for each research paper which was deemed to be reproducible in principle, we follow up with an actual attempt of reproducing the results in terms of a reproducibility study (C1.4).

- In the extended version, we are not only interested in the origins of the models used as experimental subjects, but also in their basic characteristics such as size and active maintenance span (C2.3).
- We answer the new RQ3, which analyzes whether the reproducibility of experimental results correlates to the impact of a paper. The rationale behind this is to seek evidence whether carefully dealing with reproducibility of experimental results may positively influence the impact of the underlying research.

2.1.2 Research Methodology

We conduct a systematic literature review in order to compile a list of relevant papers from which we extract the data to answer our research questions RQ1 and RQ2. Our research methodology is based on the basic principles described by Kitchenham [67]. Details of our search process and research paper selection are described in Section 2.1.2. To answer RQ3, we use the relevant papers found by the systematic literature review and how reproducible we found them in RQ1. This is used to compare reproducibility of a paper to its impact. Section 2.1.2 is dedicated to our data extraction policy, structured along a refinement of our overall research questions.

Search Process and Research Paper Selection

Scope We focus on research papers in the context of model-based development that report on the development of novel methods, techniques or algorithms for managing or processing Simulink models. Ultimately, we require that these contributions are prototypically implemented within a tool whose effectiveness has been evaluated in some form of experimental validation. Tools we consider to fall into our scope are supporting typical tasks in model-based development, such as code generation [248] or model transformation [193], clone detection [178], test generation [194] and priorization [271], model checking [274] and validation [228], model slicing [186] and fault detection [241]. On the contrary, we ignore model-based solutions using Simulink for solving a specific problem in a particular domain, such as solar panel array positioning [275], motor control [225], or wind turbine design [171].

Databases and Search Strings As illustrated in Figure 2.2, we used the digital libraries of *ACM*,[2] *IEEE*,[3] *ScienceDirect*,[4] and *dblp*[5] to obtain an initial selection of research papers for our study. These platforms are highly relevant in the field of model-based development and were used in systematic literature reviews on model-based development like [175] or [129]. By using these four different digital libraries, we are confident to capture a good snapshot of relevant papers.

[2] `https://dl.acm.org` (visited on 15/12/2025)
[3] `https://ieeexplore.ieee.org/Xplore/home.jsp` (visited on 15/12/2025)
[4] `https://www.sciencedirect.com` (visited on 15/12/2025)
[5] `https://dblp.uni-trier.de` (visited on 15/12/2025)

IEEE: ("Abstract":Simulink OR "Abstract":Stateflow) AND ("Abstract":model) AND ("Abstract":evaluat* OR "Abstract":experiment* OR "Abstract":"case study") AND ("Abstract":tool OR "Abstract":program OR "Abstract":algorithm)

ACM: [[Abstract: simulink] OR [Abstract: stateflow]] AND [[Abstract: evaluat*] OR [Abstract: experiment*] OR [Abstract: "case study"]] AND [[Abstract: tool] OR [Abstract: program] OR [Abstract: algorithm]] AND [Abstract: model]

ScienceDirect: (Simulink OR Stateflow) AND (evaluation OR evaluate OR experiment OR "case study") AND (tool OR program OR algorithm)

dblp: (Simulink | Stateflow) (model (tool | program | algorithm | method))

Figure 2.1. Digital libraries and corresponding search strings used to obtain an initial selection of research papers.

According to the scope of our study, we developed the search strings shown in Figure 2.1. We use IEEE's and ACM's search feature to select publications based on keywords in their abstracts. Some of the keywords are abbreviated using the wildcard symbol ($\star$). Since the wildcard symbol is not supported by the query engine of ScienceDirect[263], we slightly adapted these search strings for querying ScienceDirect. The same applies to dblp [263], where we also included the keyword "method" to obtain more results. To compile a contemporary and timely representation of research papers, we filtered all papers by publication date and keep those that were published between January 1st, 2015 and February 24th, 2020. With these settings, we found 625 papers on IEEE, 88 on ACM, 214 on ScienceDirect[6] and 15 on dblp.

Using the bibliography reference manager JabRef,[7] these 942 papers were first screened for clones. Then, we sorted the remaining entries alphabetically and deleted all duplicates sharing the same title. As illustrated in Figure 2.2, 912 papers remained after the elimination of duplicates.

Inclusion and Exclusion Criteria From this point onwards, the study was performed by two researchers, referred to R1 and R2 in the remainder of this paper (see Figure 2.2).

Of the 912 papers (all written in English), R1 and R2 read title and abstract to see whether they fall into our scope. Both researchers had to agree on a paper being in scope in order to include it. R1 and R2 classified 92 papers to be in scope, with an inter-rater reliability, measured in terms of Cohen's kappa coefficient [23][67], at 0.86. To foster a consistent handling, R1 and R2 classified the first 20 papers together in a joint session, and reviewed differences after 200 papers again.

[6] ScienceDirect presented an initial selection of 217 papers on their web interface, out of which 214 could be downloaded.

[7] `https://www.jabref.org` (visited on 15/12/2025)

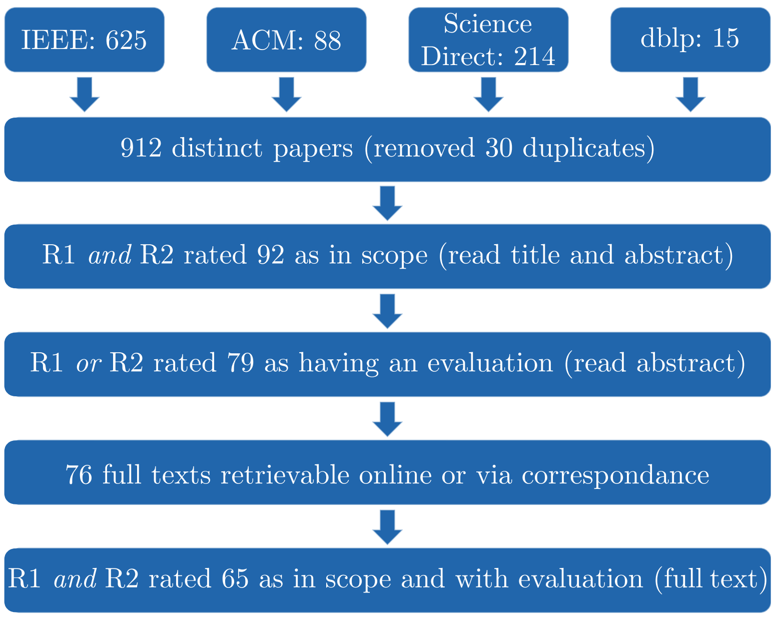

IEEE: 625
ACM: 88
Science Direct: 214
dblp: 15
912 distinct papers (removed 30 duplicates)
R1 *and* R2 rated 92 as in scope (read title and abstract)
R1 *or* R2 rated 79 as having an evaluation (read abstract)
76 full texts retrievable online or via correspondance
R1 *and* R2 rated 65 as in scope and with evaluation (full text)

Figure 2.2. Overview of the search process and research paper selection. Numbers of included research papers are shown for each step. After the initial query results obtained from the digital libraries of *ACM*, *IEEE*, *ScienceDirect* and *dblp*, the study has been performed by two researchers, referred to as R1 and R2.

Next, R1 and R2 read the abstracts and checked whether a paper mentions some form of evaluation of a presented tool. Because such hints may be only briefly mentioned in the abstract, we included papers where either R1 or R2 gave a positive vote. As a result of this step, the researchers identified 79 papers to be in scope *and* with some kind of evaluation.

We then excluded all papers for which we could not obtain the full text. Our university's subscription and the social networking site ResearchGate[8] could provide us with 45 full text papers. In addition, we found 5 papers on personal pages and obtained 28 papers in personal correspondence. We did not manage to get the full text of 3 papers in one way or the other. In sum, 76 papers remained after this step.

Finally, we read the full text to find out whether there was indeed an evaluation, as indicated in the abstract, and whether Simulink models were used in that evaluation. We excluded 10 full papers without such an evaluation and one short paper which we considered to be too unclear about their evaluation. For this last step R1 and R2 resolved all differences in classification: concerning papers were read a second time, to decide together about their inclusion or exclusion. We did this so that R1 and R2 could work with one consistent set for the data extraction. After all inclusion and exclusion steps, R1 and R2 collected 65 papers which were to be analyzed in detail in order to extract the data for answering our research questions.

[8] https://www.researchgate.net (visited on 15/12/2025)

Refinement of Research Questions and Data Extraction
In order to answer our research questions, R1 and R2 extracted data from the full text of all the 65 papers selected in the previous step. To that end, we refined our overall research questions into criteria which are supposed to be answered in a straightforward manner, typically by a classification into "yes", "no", or "unsure".

RQ1: Are the experimental results of evaluating tools supporting model-based development with Simulink reproducible?

To answer RQ1, we start with an investigation of the accessibility of models and tools, which are basic prerequisites for reproducing experimental results, followed by full reproduction studies provided these basic prerequisites are fulfilled.

Accessibility of the models. We assume that the effectiveness of a tool supporting model-based development can only be evaluated using concrete models serving as experimental subjects. These subjects, in turn, are a necessary precondition for reproducing experimental results. They should be accessible as a digital artifact for further inspection. In terms of Simulink, this means that a model should be provided as a *.mdl or *.slx file. Models that are only depicted in the paper may be incomplete, *e.g.*, due to parameters that are not shown in the main view of Simulink, sub-systems which are not shown in the paper, etc.

The aim of C1.1 is to find out, whether all models which are required for result reproduction are accessible:

C1.1: Are all models accessible?

The accessibility of models can only be checked if the paper provided us some hint of how to access them. For a given paper, we thus read the evaluation section of a paper closely, and also looked at footnotes, the bibliography as well as at the very start and end of the paper. In addition, we did a full text search for the keywords "download", "available", "http" and "www.". Next we checked whether given links indeed worked and models could be found, there. For all papers, a positive answer to C1.1 requires that each of the models used in the paper's evaluation falls into one of the following categories:

- There is a non-broken hyperlink to an online resource where the Simulink model file can be obtained from.
- There is a known model suite or benchmark comprising the model, such as the example models provided by Simulink.
- The model is taken from another, referenced research paper. In this case, we assume it to be accessible without checking the original paper.

Accessibility of the tool. Next to the models, the actual tool being presented in the research paper typically serves as the second input to reproduce the experimental results. In some cases, however, we expect that the benefits of a tool can be shown "theoretically", *i.e.*, without any need for actually executing the tool. To that end, before dealing with accessibility issues, we assess this general need in C1.2:

C1.2: Is the tool needed for the evaluation?

We read the evaluation section to understand whether there is the need to execute the tool in order to emulate the paper's evaluation. For those papers for which C1.2 is answered by "yes", we continue with C1.3. All papers answered with "no" are treated as if they provide their tool in C1.3.

Similarly to our investigation of the accessibility of models, we also assess if a paper provides access to their presented tool:

C1.3: Is the tool accessible?

In contrast to the accessibility of models, which we assume to be described mostly in the evaluation section, we expect that statements on the accessibility of a tool being presented in a given research paper may be spread across the entire paper. This means that the information could be "hidden" anywhere in the paper, without us being able to find it in a screening process. To decrease oversights, we did full text searches, for the key words "download", "available", "http" and "www.". If a tool was named in the paper, we also did full text searches for its name. A tool was deemed accessible if a non-broken link to some download option was provided. If third-party tools are being required, we expected some reference on where they can be obtained. We considered Simulink or any of its toolboxes as pre-installed and thus accessible by default.

Replicability studies. For those papers where we determined the models and tools to be publicly available, we also investigate whether the experiments are actually reproducible by us or not:

C1.4: Are the experiments reproducible?

Following the general meaning of reproducibility used throughout this paper, an experiment was deemed reproducible if its results can be reproduced by researchers different from those of the paper. To that end, all reproduction studies have been conducted by two graduate students R2 and R3[9] of our department, both of them holding a Bachelor's degree in computer science. As the reproduction studies were only done for 8 papers, we report on our experience for each of these studies from a qualitative point of view.

RQ2: From where do researchers acquire Simulink models used as experimental subjects and what are their basic characteristics?

[9] R3 was only involved for this criterion.

Next to the accessibility of models as part of RQ1, we are interested in where the researchers acquire Simulink models for the sake of experimentation and what are their basic characteristics such as size and active maintenance span. By collecting these insights, researchers in need of models for analysis or tool validation may emulate successful ways of getting models. In order to learn more about the context of a model or to get an updated version, it may be useful to contact the model creator, which motivates C2.1:

C2.1: Are all model creators mentioned in the paper?

By the term "creator" we do not necessarily mean an individual person. Instead, we consider model creation on a more abstract level, which means that a model creator could also be a company which is named in a paper or any other referenced research paper. If creators of all models were named, we answered C2.1 with "yes".

Next to the model creator, C2.2 dives more deeply into investigating a model's origin:

C2.2: From where are the models obtained?

C2.2 is one of our sub research questions which cannot be answered by our usual "yes/no/unsure scheme". Possible answers were "researchers designed model themselves"*, "generator algorithm", "mutator algorithm", "industry partner"*, "open source", "other research paper"*, "Simulink-standard example"*, "multiple" and "unknown". The categories marked with a * were also used in [129]. As opposed to us, they also used the category "none", which we did not have to consider, due to our previous exclusion steps. The category "multiple" was used whenever two or more of these domains were used in one paper. Note that even if C2.1 was answered with "no", we may still be able to answer this question. For example, if the model was acquired from a company which is not named in the paper (*e.g.*, due to a non-disclosure agreement), we may still be able to classify it as from an industry partner.

Finally, with C2.3 we give an outline about the basic characteristics of the models that were used in the experiments:

C2.3: What are the basic characteristics of the experimental models?

With this, we are interested in what kinds of models researchers use: Are the models small toy examples created in a one-shot manner for the sake of illustration, or are they bigger and actively maintained over a certain period of time?

Simulink models are block diagrams used to model dynamical systems, where computing blocks are connected by signal lines, see Figure 2.3 for a sample model. Blocks of various kinds (*e.g.*, Sum, Logic, Switch, etc.) can apply transformations on their ingoing signals, thereby producing modified outgoing signals. Inport and Outport blocks are specific blocks connecting a model with its surrounding context. Another special kind of block is the Subsystem block which can be used to modularize a Simulink model in a hierarchical manner. For more details, we kindly refer to Simulink introductions, *e.g.*, [285]. We used basic Matlab scripts to compute the following model metrics in order to characterize the models used as experimental subjects:

- number of blocks,

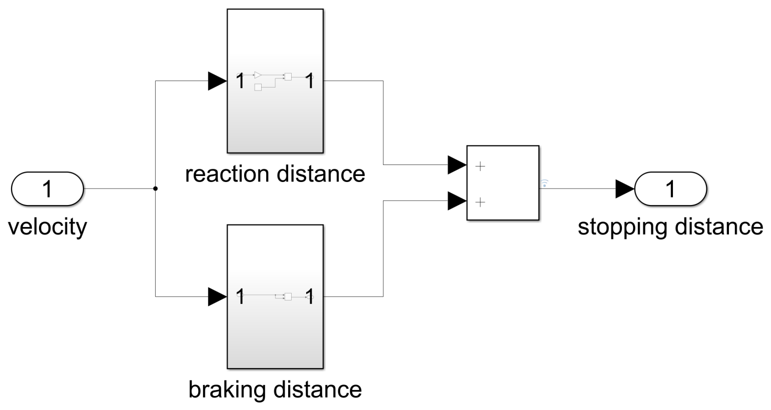


Figure 2.3. A sample Simulink model showing Inport/Outport, Add and Subsystem blocks connected by signal lines. It computes the stopping distance from a car's velocity, by summing up reaction distance and braking distance. The details of the computation of reaction distance and braking distance are abstracted from in this view.

- amount of unique block types,
- number of subsystems,
- length of active maintenance span (creation date to last save date of a model).

We also counted, how many models were provided in the reproduction packages.

RQ3: Does the reproducibility of experimental results correlate to the impact of a paper?

With this RQ, we are interested whether a paper's impact may be influenced by its handling of reproducibility. We thus investigate whether papers that rank better in terms of reproducibility (using our results of C1.1, C1.3, C1.4) have a higher relative impact than the other papers. While our analysis cannot show an actual cause and effect relationship between reproducibility and impact, there are other studies [69, 134] which let us hypothesize about a possible connection between the two.

We compute a paper's relative impact with the normalized citation score of Waltman *et al.* [140], and use a paper's citation count as the basis of this score. We collected the citation counts of each included paper from GoogleScholar[10] and ResearchGate.[11]

To strengthen our confidence in our computed value of the relative impact (see threats to validity), we compared it to the Scopus Field-Weighted Citation Impact,[12] whenever Scopus provides it for a paper. We did not use Scopus for our computation because they did not list impacts for 4 papers, and their opaque computation of the Field-Weighted Citation Impact. Finally, we group our included papers into 5 groups according to our classification

[10] https://scholar.google.com (visited on 15/12/2025)
[11] https://www.researchgate.net (visited on 15/12/2025)
[12] https://scopus.com (visited on 15/12/2025)

of C1.1/C1.3/C1.4: no reproduction package, only software provided, only models provided, both provided and reproducible. Comparisons then take place based on the average relative impact of the groups.

2.1.3 Results

In this section, we synthesize the results of our study. All paper references found, raw data extracted and calculations of results synthesized can be found in the reproduction package of this paper [327]. The package includes all the Simulink models we found during our study.

Are the experimental results of evaluating tools supporting model-based development with Simulink reproducible? (RQ1)

Table 2.1. Detailed summary of reproducibility in principle.

Criterion	Title	"yes"		"no"		"unsure"		kappa
		R1	R2	R1	R2	R1	R2	
C1.1	Are all models accessible?	13	16	52	49	0	0	0.78
C1.2	Is a tool needed?	62	58	2	2	1	5	0.58
C1.3	Is the tool accessible?	19	17	0	0	39	41	0.68

First we summarize the results for the criteria C1.1 through C1.4, before we draw our conclusions for answering the overall research question RQ1. Table 2.1 provides details for C1.1 through C1.3, while Table 2.2 shows the results for C1.4.
It can be seen that R1 and R2 generally had a high inter-rater reliability and agreed that most papers did not make their models accessible. Almost all paper's evaluations needed a tool to be executed for its evaluation. Finally we were unsure for the majority of the papers, whether they provide access to their tools.
While answering C1.3, R1 and R2 first used the additional category "no", but this produced an unsatisfactory inter-rater reliability of only 0.42. To remedy this, we revised the answers, merging the categories "no" and "unsure", acknowledging that R1 and R2 interpreted "no" and "unsure" too differently.

C1.4: Are the experiments reproducible? Table 2.2 summarizes those papers for which the models and tools were determined to be publicly available by R1 or R2, and for which R2 and R3 attempted to fully reproduce the experimental results. The reproduction studies have been conducted during October and November 2020, with two different hardware/software setups. We used a main setup running Windows 10 on a 64GB RAM, AMD Ryzen 7 3800x CPU desktop PC and MatlabR2020b. In case of a reproduction failure, we retried on a server running SuSe Leap 15 on a 1TB RAM, 4 Intel Xeon E7-4880 CPUs and Matlab R2019a. When parts of the reproduction package were inaccessible,[13] we contacted the authors to provide

[13] This means our prior designation of "reproducible in principle" by R1 or R2 was too favorable, or models or tools were no longer accessible for R2 and R3.

Table 2.2. List of papers whose experiments we examined for reproducibility.

No.	Title	reproducible?	Reference
1	A Synchronous Look at the Simulink Standard Library	no	[210]
2	Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain With SLforge	partially	[235]
3	Contract-based verification of discrete-time multi-rate Simulink models	no	[165]
4	Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models	no	[274]
5	Multi-Objective Black-Box Test Case Selection for Cost-Effectively Testing Simulation Models	partially	[233]
6	Pareto efficient multi-objective black-box test case selection for simulation-based testing	partially	[260]
7	SyLVaaS: System Level Formal Verification as a Service	partially	[173]
8	Test Suite Prioritization for Efficient Regression Testing of Model-based Automotive Software	no	[223]

us with access to models and tools for reproduction purposes. We will use the numbers listed in the first column of Table 2.2 to refer to each of the papers.

Paper No.1 provided a Docker container for all its files. Still, R2 and R3 were not able to reproduce the experiments, because they lacked knowledge of the Zélus tool. More explicit instructions in the handling of it were needed. The authors of the paper acknowledge that showing an equivalence between the Zélus blocks and Simulink blocks is hard to prove.

Paper No.2 offers a detailed documentation on how to perform the experiments, however documentation was missing in how to edit the configuration file for experimental reproduction. Both R2 and R3 tried to map the configuration parameters to those given in the paper, but the reproduction still failed for RQ2. A more detailed description would be necessary, as well as instructions on how to acquire the CyFuzz data for comparison, which was not provided by the authors.

Paper No.3 could not be reproduced because the link to the tool was not accessible at the time of conducting our reproduction studies.

Paper No.4 was not reproducible. At the time of conducting our reproduction studies we were unable to gain access to the necessary QVtrace package. Similarly no access to the author's Google Drive was granted to us.

Paper No.5 offers all software components and models including a minimal documentation. While trying to reproduce the experiments, we received numerous warnings and errors and were able to generate simulation times for only three out of four models. It was unclear how to deal with the simulation times, since the paper and documentation do not offer explicit instructions on how to aggregate the results such that they are comparable to those offered by the authors.

Paper No.6 is a revised version of paper No.5. It adds two more models to the experimental evaluation. As opposed to the first version of the paper, the ReadMe provided claims that

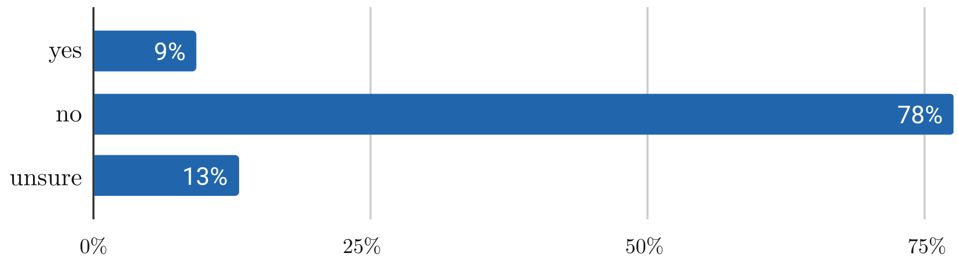


Figure 2.4. Are studies of model-based development reproducible in principle?

scripts to execute the experiments are now being provided. However, we could only find one of the scripts, which was not executable. Thus, we were able to get simulation times for four out of six models, which were again not comparable with the authors' data.

Paper No.7's online tool described in the paper was not accessible. The authors instead provided us with a similar offline Docker container. Even though the documention was very detailed, the important parameter τ was missing. Furthermore, models mentioned in the paper were not accessible to us anymore. Nevertheless, we were able to generate models with the tool, concluding that these experiments were the closest to reproduction.

Paper No.8 was not reproducible, as implementation files were missing. As their model "Gearbox" and the Reactis tool are publicly available, only the very first step of their experiments could be reproduced.

Aggregation and summary of the results. To answer RQ1, we first assess whether the experiments described in the studied papers are reproducible in principle. Therefore, we combine C1.1 and the revised answers of C1.3. For those papers where there was no tool needed (C1.2), C1.3 was classified as "yes". The formula we used is "If C1.1 = 'no' then 'no' else C1.3". This way, on average, 6 (R1: 5, R2: 7) papers have been classified as reproducible in principle, 50.5 (R1: 52, R2: 49) as not reproducible, and 8.5 (R1: 8, R2: 9) for which we were unsure (see Figure 2.4), with Cohen's kappa of 0.67. In sum, 8 papers were classified to be reproducible in principle by at least one of the researchers.
We were not able to fully reproduce any of the experiments of those 8 papers, and achieved partial reproducibility for four of them. Three tools were not completely available to us due to denied access, timeouts and a missing implementation. One paper could not be reproduced due to incomplete documentation. Four software setups were closely examined and principally functional, but the experiments could not be fully reproduced due to incomplete documentation, errors or broken links to the models used as experimental subjects.

***RQ 1:** Are the experimental results of evaluating tools supporting model-based development with Simulink reproducible?*
We found 9% of the examined papers to be reproducible in principle as software and models were provided. However, none of the experiments could be fully reproduced, while we achieved partial reproduction for 6%. For 78% of the examined papers, either the tool or the models used as experimental subjects were not accessible, and we were not able to determine the principle reproducibility of the experiments presented in 13% of the investigated papers as we were unsure about the accessibility of tools.

From where do researchers acquire Simulink models used as experimental subjects and what are their basic characteristics? (RQ2)
C2.1 Are all model creators mentioned in the paper? As can be seen in Figure 2.5, Of the 65 papers investigated in detail, on average, 44 (R1: 43, R2: 45) papers mention the creators of all models. On the contrary, no such information could be found for an average of 20.5 (R1 22, R2 19) papers. Finally, there was one paper for which R2 was not sure, leading to an average value of 0.5 (R1: 0, R2: 1) for this category. In sum, this question was answered with an inter-rater reliability of 0.79.

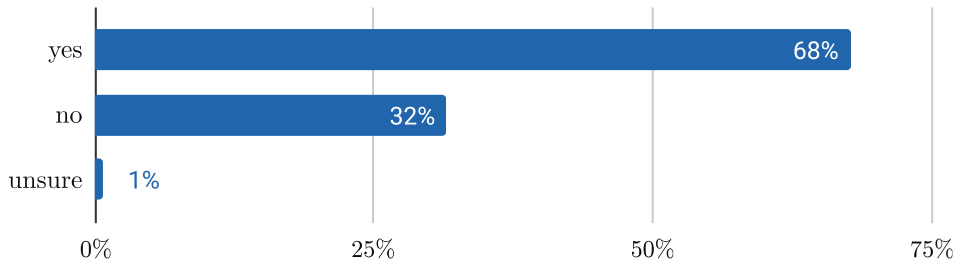


Figure 2.5. C2.1: Are all model creators mentioned in the paper?

C2.2 From where are the models obtained? As shown in Figure 2.6, there is some variety for the model's origins. Only 3% used open source models, 8% used models included in Simulink or one of its toolboxes, 12% cited other papers, 13% built their own models, and 18% obtained models from industry partners. A quarter of all papers used models coming from two or more different sources. For 19% of the papers, we could not figure out where the models come from. This mostly coincides with those papers where we answered "no" in C2.1. For some papers, we were able to classify C2.2, even though we answered C2.1 with "no". E.g. we classified the origin of a model of [190] as "industry partner" based on the statement "a real-world electrical engine control system of a hybrid car", even though no specific information about this partner was given. C2.2 was answered with Cohen's kappa of 0.68.

An interesting yet partially expected perspective arises from combining C2.2 and C1.1. None of the models obtained from an "industry partner" are accessible. Three papers which we classified as "multiple" in C2.2 did provide industrial models though: [274] provides models from a "major aerospace and defense company" (name not revealed due to a non-disclosure agreement), while [233] and [260] use an open source model of electro-mechanical

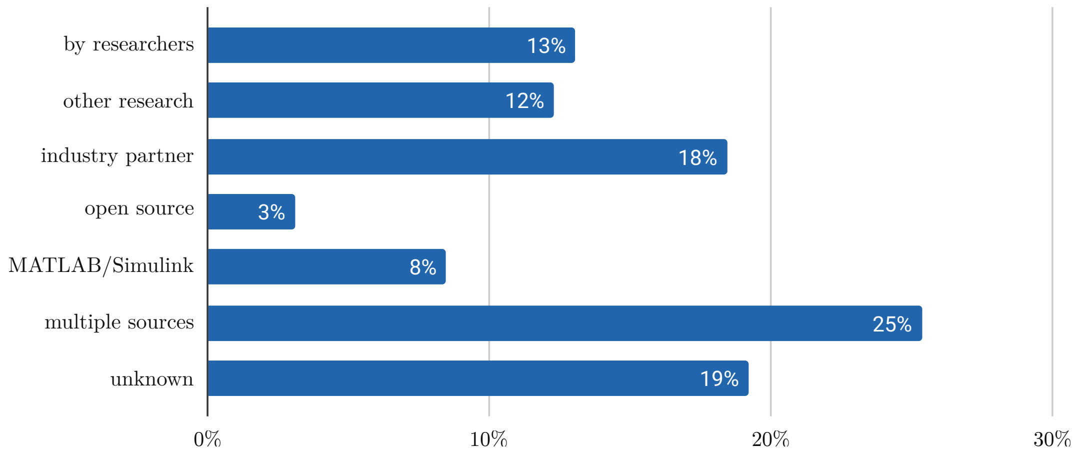


Figure 2.6. C2.2: From where are the models obtained?

braking provided by Bosch in [179]. Finally, [208] and [244] use models for an advanced driver assistance system by Daimler [374], that can be inspected on the project website.[14]

C2.3 What are the basic characteristics of experimental models? Table 2.3 lists basic characteristics of the models used as experimental subjects in the papers investigated by our study. For all of the characteristics, the standard deviation was high, and the distributions of the measures are right-skewed (median smaller than mean). Most of the papers used only a few models for their experiments, as the median is at only three. We found the models themselves to be not only toy examples, as there were 26 subsystems and 236 blocks per model in the median. A further indication of this is that the median model uses 12 different block types. Figure 2.7 gives an impression of the 25 most frequently used blocks. The three most frequently used kinds of blocks are the Inport, Outport and Subsystem blocks; these are used for the sake of modularizing a model into different parts. All this shows some degree of sophistication in the models. Certainly, a “degree of sophistication” is not objectively measurable, but talking to Simulink experts in private conversation, they reported typical models in industry often having 1000-10000 blocks, most of the models we found in this study are smaller. Finally, we determined the maintenance spans of the models, which has the highest degree of deviation. There are many models with a maintenance span of “zero” days, but we also observe models with a maintenance spans of multiple years.

[14] https://www.se-rwth.de/materials/cncviewscasestudy (visited on 15/12/2025)

Table 2.3. Basic characteristics of the models used as experimental subjects in the papers which were found by our systematic literature search.

Category	Min	Max	Mean	Median	Std.-Dev.
#Models per paper	1.0	215.0	51.5	3.0	81.4
#Blocks per model	2.0	3276.0	305.5	236.0	364.6
#Subsystems per model	1.0	135.0	26.1	26.0	18.7
#Unique blocks per model	2.0	55.0	16.1	12.0	7.0
Maintenance span in days	0.0	7678.0	554.8	0.0	1076.5

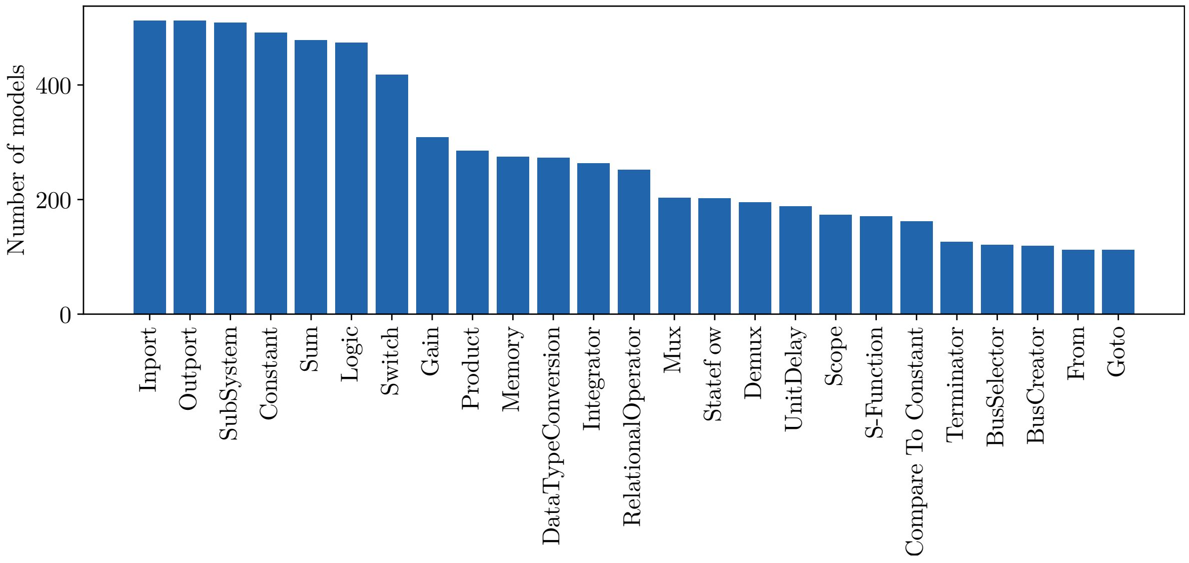


Figure 2.7. Overview of the most frequently used kinds blocks in the Simulink models. The horizontal axis lists Simulink block types, the vertical axis shows how many models did use this type of block.

RQ 2: *From where do researchers acquire Simulink models used as experimental subjects and what are their basic characteristics?*
A wide variety of sources is used. 25% used multiple sources, another 25% used models of their own or from other researchers, 18% used models by an industry partner. 8% used models of Simulink or a toolbox and only 3% used open source models. We could not determine 19% of the models' origins. A basic analysis of the models' characteristics showed them having a degree of sophistication containing 236 blocks, 26 subsystems, and 12 different block types in the median. On the contrary, the median maintenance span is at zero days, which suggests that most of the models have been created in a one-shot manner for the sake of illustration.

Does the reproducibility of experimental results correlate to the impact of a paper? (RQ3)

To compute the relative impact of a paper, we collected the citation counts on GoogleScholar and ResearchGate. GoogleScholar provided mostly higher counts of 8.6 average citations versus 5.8 on ResearchGate[15]. Because of the complete citation record and the generally higher values, we used the results of GoogleScholar for further computations of the relative impact. A histogram of the relative impact of the papers is shown in Figure 2.8.

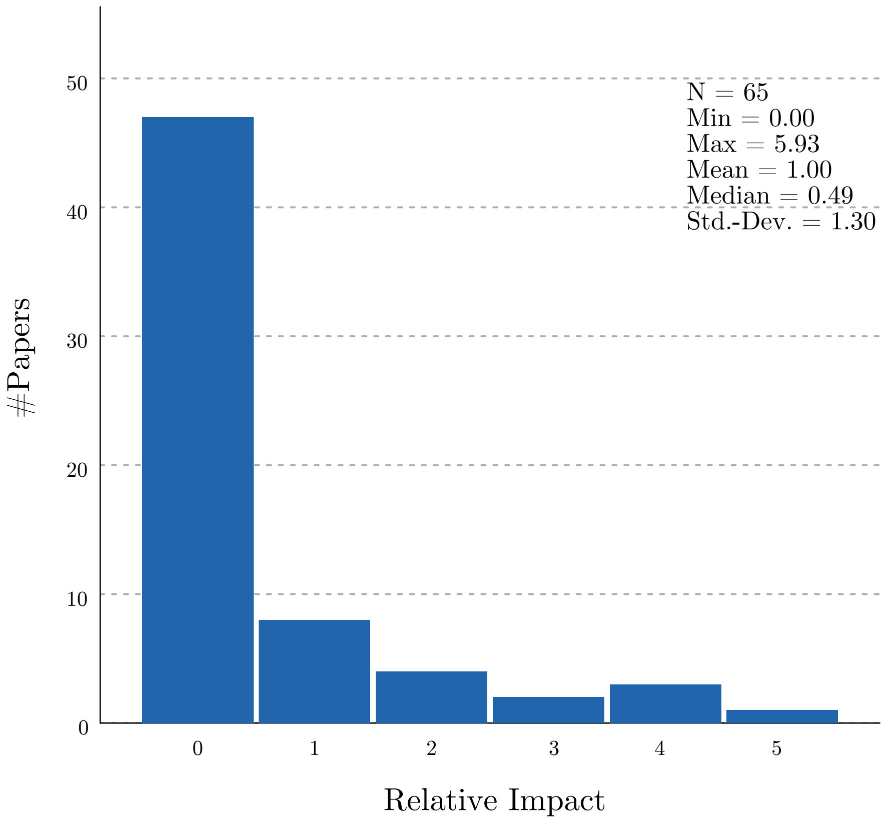


Figure 2.8. Histogram of the relative impact of the papers. An average paper has a relative impact of 1.00, which means that it is cited as often as the average of its peers. One paper (the maximum) is cited 5.93 as often as its peers and some (the minima) were never cited and thus have a relative impact of 0.00.

We compared our computed relative impacts with the Scopus Field-Weighted Citation Impact by computing the correlation between the two impacts with SPSS.[16] Spearman's rank correlation coefficient is 0.838 with significance $p < 0.001$, and Kendall's rank correlation coefficient is 0.668 with significance $p < 0.001$. We also applied Wilcoxon's signed-rank test which indicates that the median of differences between our impact and the Scopus impact is the same, with significance $p = 0.074 > \alpha = 0.05$. Finally the average value of our relative impact is 1.0 vs. 1.08 for the Scopus Field-Weighted Citation. With these similarities between the two measures, we concluded that our relative impact can be used for further analysis.

Next, we grouped the papers according to our findings of C1.1, C1.3 into the 4 groups: "nothing provided", "only models provided", "only software provided", "both provided". If at least one researcher found the models were provided, or at least one researcher found

[15] Three of the 65 papers could not be found on ResearchGate.

[16] `https://www.ibm.com/products/spss-statistics` (visited on 15/12/2025)

Table 2.4. Average relative impacts grouped by our results of C1.1 and C1.3.

Group	#Papers	Average relative impact
Nothing provided	27	0.590
Only models provided	9	0.739
Only software provided	21	1.369
Both provided	8	1.710

the software was provided or not necessary, the respective group is used. Note that if both researchers were "unsure", this was not the case. The average relative impacts are presented in Table 2.4. The 4 papers achieving partial reproducibility (part of "both provided") even got an average impact of 2.072. The average relative impacts of the 4 groups are growing monotonically as listed in the table. The findings signal a positive correlation between a paper's reproductive quality and its relative impact. Papers that only provided their software scored higher than papers that only provided their models. This does not imply a cause and effect relationship, though (see Section 2.1.4).

RQ 3: *Does the reproducibility of experimental results correlate to the impact of a paper?* Grouped by our findings of C1.1 and C1.3, we found that groups of papers with higher reproducibility achieved a higher relative impact. Papers we could partially reproduce also got the highest relative impacts.

2.1.4 Threats to Validity

There are several reasons why the results of this study may not be representative, the major threats to validity as well as our countermeasures to mitigate these threats are discussed in the remainder of this section.

Our initial paper selection is based on selected databases and search strings. The initial query result may not include all the papers being relevant w.r.t. our scope. We tried to remedy this threat by using four different yet well-known digital libraries, which is sufficient according to [92] and [199] and using wild-carded and broad keywords in the search string.

For the first two inclusion/exclusion steps, we only considered titles and abstracts of the papers. If papers do not describe their topic and evaluation here, they could have been missed.

It turned out to be more difficult than originally expected to find out whether a paper provides a reproduction package or not. One reason for this is that just scanning the full text of a paper for occurrences of Simulink or the name of the tool is not very useful here since there are dozens of matches scattered all over the paper. In fact, almost every sentence could "hide" a valuable piece of information. We tried to remedy this problem by searching the *.pdf files for the key words mentioned in Section 2.1.2. We also merged our answers of "no" and "unsure" for C1.3 in reflection of this problem.

We were very strict in rating the accessibility, *i.e.*, we expected not to have to contact a paper's authors for getting model or tool access. This may have lowered the number of papers, we deemed to be reproducible in principle.

More generally, the data collection process was done by two researchers, each of which may have overlooked important information or misinterpreted a concrete criterion. We tried to mitigate these issues through intermediate discussions. Furthermore, we calculated Cohen's kappa coefficient to better estimate the reliability of our extracted data and synthesized results.

Furthermore, the reproduction attempt of the experiments of 8 papers in C1.4 was only conducted by two researchers. The researchers did not have complete expert knowledge in the fields of the analyzed papers, which could have caused difficulties to reproduce experiments. This together with our strict grading of reproducibility may have lead to a lower number of papers being reproducible in principle (only 8 of 65) and papers we could fully reproduce (none of the 8).
We did not investigate the paper's Simulink versions or hardware setups for our assessment of principal reproducibility. This is because in many cases newer versions of Simulink or our own hardware setups would produce equivalent results.

Our methodology section does not present a separate quality assessment which is typical in systematic literature review studies [120]. Thus, our results could be different if only a subset of high quality papers, *e.g.*, those published in the most prestigious publication outlets, would be considered. Nonetheless, a rudimentary quality assessment (paper's language, experimental evaluation instead of "blatant assertion" [49]) was done in our inclusion/exclusion process.

For the analysis of the Simulink models in C2.3, we used self-written Matlab scripts, which could be faulty. We especially cannot rule out that the maintenance span was computed correct for some models, as many models had a maintenance span of 0 days. This could naturally occur by a very short lived model or an automatic script creating and saving a model instantaneously. Another possibility is, that the date feature of Simulink is buggy for some models.

As already indicated, there is a threat to the conclusion validity for answering RQ3 since there is not necessarily a cause and effect relationship between reproducibility of experimental results and the relative impact of a paper. Another possible cause could be the reputation or impact factor of the publication venue. If this is the case, however, then our results may point to higher reproduction standards for these venues. Moreover, our computed relative impact score is based on only 65 papers grouped by their publication year into 5 peer groups of size 9, 20, 15, 15 and 6. This is why we compared our relative impact with the Scopus metric for average, correlation and distribution and found it to be highly similar. Another mitigating factor is that the papers were manually selected in our systemic literature review. This ensures comparing papers only with highly relevant peers in each peer group.

2.1.5 Discussion

Limited accessibility of both models and tools. Although generally accepted, the FAIR guiding principles of good scientific practice are hardly adopted by current research on tools for model-based development with Simulink. From the 65 papers which have been selected for an in-depth investigation in our systematic literature review, we found that only

22% of the models and 31% of the tools required for reproducing experimental results are accessible. Thus, future research that builds on published results, such as larger user or field studies, studies comparing their results with established results, etc., are hardly possible, which ultimately limits scientific progress in general.

Difficulties regarding reproducibility. We found none of 8 thoroughly examined papers to be fully reproducible. This was largely caused due to insufficient documentation on the experiment setups, parameters and tools, or missing parts of implementations, tools or models. We suggest providing Docker containers or similar for ease of experimental evaluation. These can come pre-installed and pre-configured. This way, the reproducor simply has to download the container and start a script.

Reproducibility and relative impact. We can confirm the finding, of papers publishing their data being cited more often [69, 134, 221]. Our results further show, that publishing software or both data and software does have a higher correlation with citation counts, than just publishing the experimental datasets.

Open source mindset rarely adopted. One general problem is that the open source mindset seems to be rarely adopted in the context of model-based development. Only 3% of the papers considered by our study obtained all of their models from open source projects. On the contrary, 18% of the studied papers obtain the models used as experimental subjects from industry partners, the accessibility of these models is severely limited by confidentiality agreements.

Selected remarks from other papers. These quantitative findings are also confirmed by several authors of the papers we investigated during out study. We noticed a number of remarks w.r.t. the availability of Simulink models for evaluation purposes. Statements like "To the best of our knowledge, there are no open benchmarks based on real implementation models. We have been provided with two implementation models developed by industry. However, the details of these models cannot be open."[283]; "Crucial to our main study, we planned to work on real industrial data (this is an obstacle for most studies due to proprietary intellectual property concerns)."[208]; "[...] most public domain Stateflows are small examples created for the purpose of training and are not representative of the models developed in industry."[267]; or "Such benchmarking suites are currently unavailable [...] and do not adequately relate to real world models of interest in industrial applications."[228] discuss the problem of obtaining real-world yet freely accessible models from industry. Other statements such as "[...] as most of Simulink models [...] either lack open resources or contain a small-scale of blocks."[218] or "[...] no study of existing Simulink models is available [...]."[234, 235] discuss the lack of accessible Simulink models in general.

Reflection of our own experience. In addition, the findings reflect our own experience when developing several research prototypes supporting model management tasks, *e.g.*, in terms of the SiDiff/SiLift project [110, 111]. Likewise, we made similar observations in the

SimuComp project. Companies want to save the intellectual property and do not want their (unobfuscated) models to be published.

As opposed to the lack of availability of models, we do not have any reasonable explanation for the limited accessibility of tools. Most of the tools presented in research papers are not meant to be integrated into productive development environments directly, but they merely serve as experimental research prototypes which should not be affected by confidentiality agreements or license restrictions.

Suggestions based on the lessons learnt from our study. While the aforementioned problems are largely a matter of fact and cannot be changed in a short-term perspective, we believe that researchers could do a much better job in sharing their experimental subjects. Interestingly, 12% of the studies obtain their experimental subjects from other papers, and 13% of the papers state that such models have been created by the authors themselves. Making these models accessible is largely a matter of providing adequate descriptions.

However, such descriptions are not always easy to find within the research papers which we considered in our study. Often, we could not find online resources for models or software. It should be made clear where to find reproduction packages. In some cases a link to the project's website was provided, but we couldn't find the models there. To prevent this, we suggest direct links downloadable files or very prominent links on the website. The web resource's language should match the paper's language: *e.g.*, the project site of [256] is in German. Four papers referenced pages that did not exist anymore, *e.g.*, a private Dropbox[17] account. These issues can be easily addressed by a more thorough archiving of a paper's reproduction package.

We also suggest to name or cite creators of the models, so they can be contacted for a current version or context data of the model. In this respect, the results of our study are rather promising. After all, model creators have been mentioned in 68% of the studied papers, even if the models themselves were not accessible for a considerable amount of these cases.

Towards larger collections of Simulink models. Our study not only reveals the severity of the problem, it may also be considered as a source for getting information about publicly available models and tools. In fact, we found a number of papers that did publish their models. This includes models with a degree of sophistication, *e.g.*, models with more than a few hundred blocks. These models could be reused, updated or upgraded in other studies. We provide all digital artifacts that were produced in this work (.bibtex files of all papers found, exported spread sheets and retrieved models or paper references) online for download at [327]. Altogether we downloaded 517 Simulink models. We also found 32 referenced papers where models were drawn from. These models could be used by other researchers in their evaluation. Further initiatives of providing a corpus of publicly available models, including a recent collection of Simulink models, will be discussed in the next section.

[17] `https://www.dropbox.com` (visited on 15/12/2025)

2.1.6 Related Work

The only related secondary study we are aware of has been conducted by Elberzhager *et al.* [129] in 2013. They conducted a systematic mapping study in which they analyzed papers about quality assurance of Simulink models. This is a sub-scope of our inclusion criteria, see Section 2.1.2. One research question of them was "How are the approaches evaluated?". They reviewed where the models in an evaluation come from and categorized them into "industry example", "Matlab example", "own example", "literature example" and "none". We include more categories, see Section 2.1.2, apart from "none". All papers that would fall in their "none"-category were excluded by us beforehand. Compared to their findings, we categorized 2 papers using open source models, one with a generator algorithm and 16 with multiple domains. Furthermore we found 11 papers, where the domain was not specified at all. They also commented on our RQ1: "In addition, examples from industry are sometimes only described, but not shown in detail for reasons of confidentiality."

The lack of publicly available Simulink models inspired the SLforge project to build the only large-scale collection of public Simulink models [235, 236] known to us. To date, however, this corpus has been only used by the same researchers in order to evaluate different strategies for testing the Simulink tool environment itself (see, *e.g.*, [291]). Another interesting approach was used by Sanchez *et al.* [282]. They used Google BigQuery[18] to find a sample of the largest available Simulink models on GitHub. In sum, they downloaded 70 Simulink models larger than 1MB from GitHub. Some authors created datasets of Simulink models as benchmarks. For example, Bourbouh et al. [209] compiled a set of 77 Stateflow models to demonstrate the effectiveness of their tool. Another benchmark of Simulink models was created as part of the Applied Verification for Continuous and Hybrid Systems (ARCH) workshop [262]. Regarding works describing characteristics of Simulink models, Dajsuren et al. [127, 167] reported coupling and cohesion metrics they found in ten industrial Simulink models. They measured the interrelation of subsystems as well as the interrelation of blocks within subsystems.

In a different context focusing on UML models only, Hebig *et al.* [189] have systematically mined GitHub projects to answer the question when UML models, if used, are created and updated throughout the lifecycle of a project. A similar yet even more restricted study on the usage of UML models developed in Enterprise Architect has been conducted by Langer *et al.* [151].

Apart from individual studies, there is an increasing community effort towards the adoption of open science principles within the field of software and systems engineering. One of the goals of such efforts is to create the basis for reproducing experimental results. Most notably, a number of ACM conferences and journals have established formal review processes in order to assess the quality of digital artifacts associated with a scientific paper, according to ACM's "Artifact Review and Badging" policy[19]. Artifacts may receive three different kinds of badges, referred to as "Artifacts Evaluated", "Artifacts Available", and "Results Validated". In terms of our study, we investigated the accessibility of digital artifacts in terms of C1.1 and C1.3, which is an essential prerequisite for receiving an "Artifacts

[18] https://cloud.google.com/bigquery (visited on 15/12/2025)

[19] https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 15/12/2025)

Available” badge. The notion of reproducibility used in terms of this paper and particularly addressed in C1.4 is one of the two possible forms of how experimental results may receive a “Results Validated” badge. As a natural side-effect of conducting our reproduction studies, we also investigated the documentation, completeness, consistency and excercisability of artifacts, which are the requirements for receiving an “Artifacts Evaluated” badge.

2.1.7 Conclusion

In this paper, we investigated to which degree the principles of good scientific practice are adopted by current research on tools for model-based development, focusing on tools supporting Simulink. To that end, we conducted a systematic literature review and analyzed a set of 65 relevant papers on how they deal with the accessibility of experimental reproduction packages.

We found that only 31% of the tools and 22% of the models used as experimental subjects are accessible. Given that both artifacts are needed for a reproduction study, only 9% of the tool evaluations presented in the examined papers can be classified to be reproducible in principle. We found none of those papers to be fully reproducible and only 6% of them partially reproducible. Moreover, only a minor fraction of the models is obtained from open source projects, but some of those open source models show a degree of sophistication and could be useful for other experimental evaluations. Altogether, we see this as an alarming signal w.r.t. making scientific progress on better tool support for model-based development processes centered around Simulink. Giving access to the models and tools also could potentially result in a higher impact in the scientific community – this may serve as another motivation to give more care to reproducibility. While both tool and models are essential prerequisites for reproduction and reproducibility studies, the latter may also serve as experimental subjects for evaluating other tools. In this regard, our study may serve as a source for getting information about publicly available models. Other researchers in this field have even started to curate and analyze a much larger corpus of Simulink models [236, 1]. Open source models were found to be highly diverse in almost all metrics applied with some being complex enough to be representative of industry models.

↪ Thesis Roadmap

2.2 ScoutSL: An Open-source Simulink Search Engine

Authors: Sohil Lal Shrestha, Alexander Boll, Timo Kehrer, and Christoph Csallner.

Published in: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, 2023.

Abstract: Simulink is one of the most widely used modeling languages in safety-critical industries. Most models created in industrial settings are valuable intellectual property to their companies and are thus often not publicly available. But to study Simulink software engineering processes or to develop new Simulink tools, access to models with relevant properties is vital for researchers. We conducted a community survey to find out what kind of models and model metrics are of interest to researchers. With these results, we implemented ScoutSL (`http://scoutsl.net`), a tool that gives researchers easy online access to over 100k open-source Simulink models from which they can select a subset according to their

needs. A short video demonstration is available online at `https://youtu.be/HwsHL8LrVCM`.

2.2.1 Introduction

Searching for Simulink models presents challenges due to the absence of a convenient method for finding such models beyond text-based searches. Traditional textual programming language search attributes such as lines of code do not apply to Simulink models, which are developed via graphical block diagrams. So the requisite search attributes for Simulink models are not adequately addressed.

Despite the proliferation of open-source repositories that have accelerated empirical study of code [34, 94], there is a dearth of such studies on MATLAB/Simulink. This can be attributed to the lack of easily accessible model corpora and user-friendly tools that cater to novice users. Researchers have encountered difficulties in discovering third-party Simulink models suitable for utilization in their studies, particularly for stress testing their developed tools or validating the generalizability of novel techniques they are attempting to address. The absence of an easily accessible and user-friendly tool remains the primary hindrance [127, 307, 308, 326, 340, 353, 356].

Popular code hosting platforms such as GitHub and GitLab lack the capability to filter attributes specific to Simulink models, and the identification of Simulink projects is challenging as these platforms do not label projects with Simulink as a programming language. Moreover, utilizing the APIs of these platforms for research purposes is time-consuming due to API rate limits. For instance, GitHub's API restricts authenticated users to 30 search requests per minute, yielding 1k results per request and 5k other requests per hour. Considering that GitHub currently hosts over 330 million repositories, obtaining results, excluding downloading and further analysis, would require at least 330k requests (around 180 hours) [358, 377].

Few existing model-based search tools, like MAR [334], require a deep understanding of the metamodel, while others, such as ModelMine [303], offer a user-friendly search engine but rely on the GitHub API, which inherently imposes limitations on the number of search results it can retrieve. Furthermore, GitHub is not the sole source of Simulink projects. Simulink vendor MathWorks also provides a platform, which serves as a repository for community-developed projects.

Recent efforts on developing large collections of Simulink models have focused on carefully curating corpora of Simulink models manually [236] and later automatically [337] and maintaining metadata of commonly used attributes. Such corpora are either maintained in non-permanent locations though or packaged as a single non-divisible set, making them difficult to sample [291, 337]. Downloading a large corpus to sample a small subset of models is also often inconvenient.

To gain insight into attributes of Simulink models that are of interest to the research community, we conducted a survey involving researchers. The survey confirmed their struggles in getting suitable (*e.g.*, size, publishable) models for their research as seen in Figure 2.10. Consequently, we developed a web-based search tool that allows users to easily sample models. Our tool, ScoutSL, expands upon the existing SLNET [337] and

EvoSL [10] infrastructure to extract Simulink project attributes, collect model metrics and compute derived metrics. The tool offers advanced fine-grained filtering attributes, enabling users to efficiently sample desired models. We have indexed over 18k projects containing more than 100k Simulink models. To the best of our knowledge, ScoutSL is the first tool specifically designed for searching Simulink projects and models, offering filtering attributes not available through other search engines. To summarize, the paper makes the following major contributions.

- Our survey results show that researchers often struggle to get relevant models for their research and would likely benefit from a Simulink search engine.
- We developed a Simulink search engine deployed in a tool called ScoutSL whose search interface and ranking scheme are based on survey responses.
- The tool and all artifacts are open-source [354, 355].
- The search engine is accessible through its web component available online at `http://scoutsl.net`.

2.2.2 Background: Simulink, SLNET, and EvoSL

Simulink is a cyber-physical system (CPS) design and simulation tool that is a de-facto standard in many safety-critical industries. Engineers design a CPS as a model that contains interconnected blocks, where each block may accept data, perform some operation on the data, and transmit its output to other blocks, as depicted in Figure 2.9. Simulink provides an extensive library of blocks and toolboxes to design and simulate complex multi-domain systems.

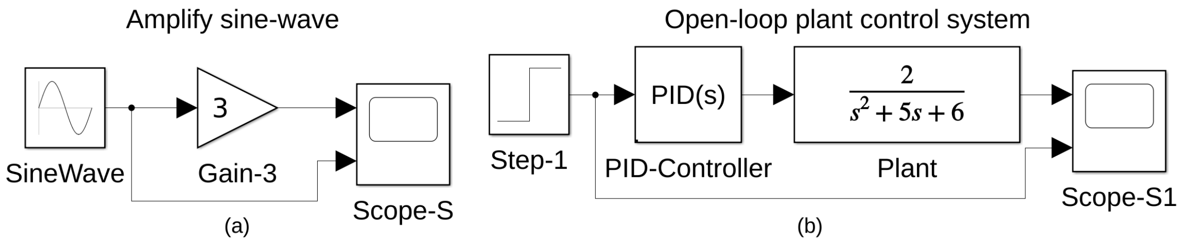


Figure 2.9. Two tiny example Simulink models.

To enable empirical studies on Simulink models, researchers have curated large corpora of open-source Simulink models. The most extensive corpus available to date is SLNET [337], which contains Simulink models from two popular hosting sites. SLNET has 3k Simulink projects with their 8k Simulink models (excluding library and test harnesses), collectively featuring over 1M blocks. Boll *et al.* [1] confirmed large open source corpora to be suitable for empirical research. SLNET is complemented by mining and metric tools. However, as SLNET primarily consists of Simulink model snapshots, it does not support evolution studies.

To address this issue, EvoSL [10] extended SLNET-Miner and curated Simulink repositories from GitHub. EvoSL consists of 924 projects with over 140k default-branch commits. SLNET and EvoSL are self-contained and redistributable.

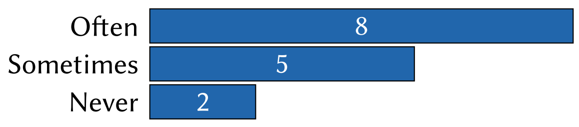


Figure 2.10. Responses to “Do you have difficulties finding adequate Simulink models or projects for your research?”

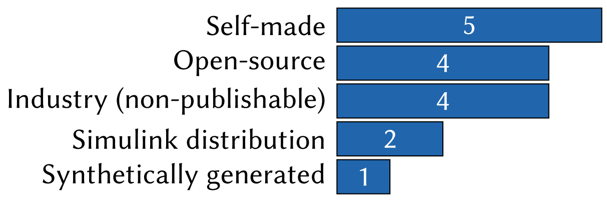


Figure 2.11. Responses to “Where do you usually obtain your Simulink artifacts from?”

2.2.3 Survey of Simulink Users

We aimed to assess the potential need for a Simulink model search engine by asking Simulink researchers. We then used the survey results to develop ScoutSL.

From a literature review of Boll *et al.* [2] we extracted 215 academic papers’ co-authors that report on Simulink tools and their empirical evaluation. In July and August 2022 we invited them to our Google Forms based anonymized online survey. 16 researchers participated in our survey, from which we discarded one participant (who responded “not applicable” to every question), leaving 15 participants. While all questions and responses are available online [354], we provide a brief summary of the questions and responses in the sequel.

In our first question of the survey, we asked the researchers, “what is the purpose of Simulink models in your research?” Of 15 participants, six reported that their main use-case for Simulink models was tool evaluation. Other use-cases like scalability evaluation, performance evaluation, model co-simulation, industrial process modelling, prototyping, test automation, testing, verification, code generation optimization, compilation, model deployment, and reproduction were mentioned by one participant each. All the following questions and their detailed results are shown in Figures 2.10 to 2.15.

Over 85% of participants (13/15) faced difficulties finding appropriate models for their research projects (see Figure 2.10). When it came to acquiring models, one third of the participants created their own Simulink artifacts, followed by using closed-source and open-

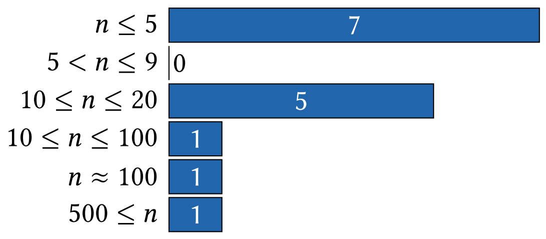


Figure 2.12. “How many models would you need for your typical research project?”

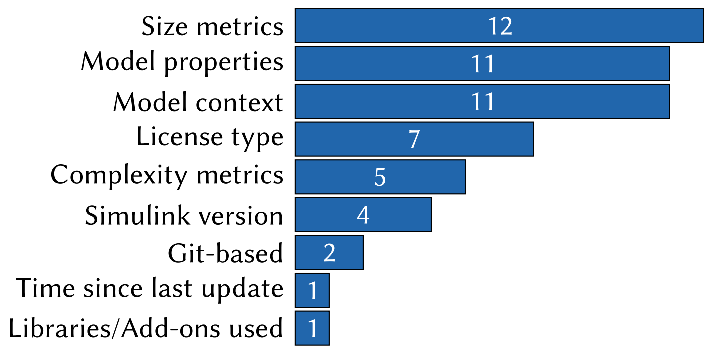


Figure 2.13. Responses to "What are Simulink model metrics that are relevant for your research?"

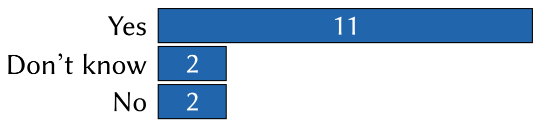


Figure 2.14. Responses to "We collected 9,117 open-source models from GitHub. Intuitively, do you think this collection can provide you with suitable Simulink models for your research?"

source models, see Figure 2.11. Figure 2.12 illustrates that the majority of participants said they require 20 or fewer models for their research—the unconventional ranges follow the responses. Figure 2.13 breaks down the model metrics of interest reported by the participants.

In response to our questions regarding the adoption of open-source Simulink models, participants showed overwhelming support for both potential usage of the dataset (see Figure 2.14) and the need for a search engine (see Figure 2.15).

2.2.4 Tool Architecture

Figure 2.16 illustrates ScoutSL's architecture, which consists of two main components: An (offline) mining component and an (online) web application component. The miner retrieves Simulink projects from the repository hosting sites and stores project, model, and commit metrics in a SQLite database.

An intermediate component queries the SQLite database, computes derived attributes such as a model's code generation capability, and calculates a "relevance" project score. A subset[20] of the primary and derived attributes are then stored in a cloud-hosted NoSQL database, as NoSQL databases typically have flexible data models and scale horizontally. The online web interface of ScoutSL facilitates user searches for Simulink projects, allowing filtering based on various attributes.

[20] Due to unclear project licenses not all SQLite data are exposed.

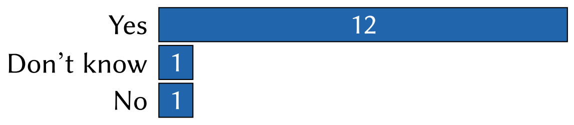


Figure 2.15. Responses to "Would you use ScoutSL for your research, in the future?"

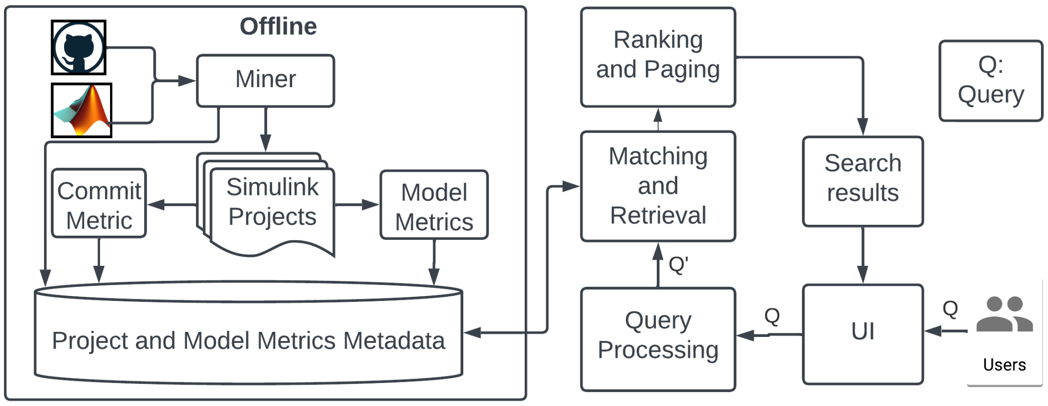


Figure 2.16. Architecture of the ScoutSL tool.

2.2.5 Mining Component

To mine from GitHub and MATLAB Central we use the existing SLNET [337] and EvoSL [10] infrastructure. Unlike SLNET or EvoSL, our focus is primarily on curating a comprehensive database of publicly available Simulink projects, and thus we do not prioritize the analysis of license files as the goal is to allow users to sample from all available open-source Simulink models. Our search yielded 18k projects comprising 109k Simulink models having 15M+ blocks ($\geq 15\times$ SLNET).

GitHub

Our extension of SLNET-Miner efficiently manages GitHub's API rate limits. Initially, we query for projects created within a specific time frame, such as *"q=simulink&created:2008-01-01..2009-01-01"*. To exhaustively search for projects using the GitHub API, we employ a divide-and-conquer strategy when the query returns 1k results. We split the time interval in half until the number of results returned is less than 1k.

From the query results we then iteratively download each project and check if it contains a Simulink model by checking for files with MDL or SLX extensions. We extract 80 attributes [354] from the projects' metadata, commits, issues, and pull requests. We only include each GitHub project's commits from its default-branch. Over a one-month period we thereby downloaded some 14k Simulink projects. To mitigate redundancy we currently do not collect metrics from forked projects (but plan to add this in the future).

MATLAB Central

As SLNET we parse MATLAB Central's RSS feed (which does not impose any parsing restrictions), but the feed does not offer a structured method for downloading projects. We extended SLNET-Miner to enhance its heuristic for constructing download links for MATLAB Central projects. Despite these improvements, 7.9k of 46k MATLAB Central projects remained inaccessible for automated download.

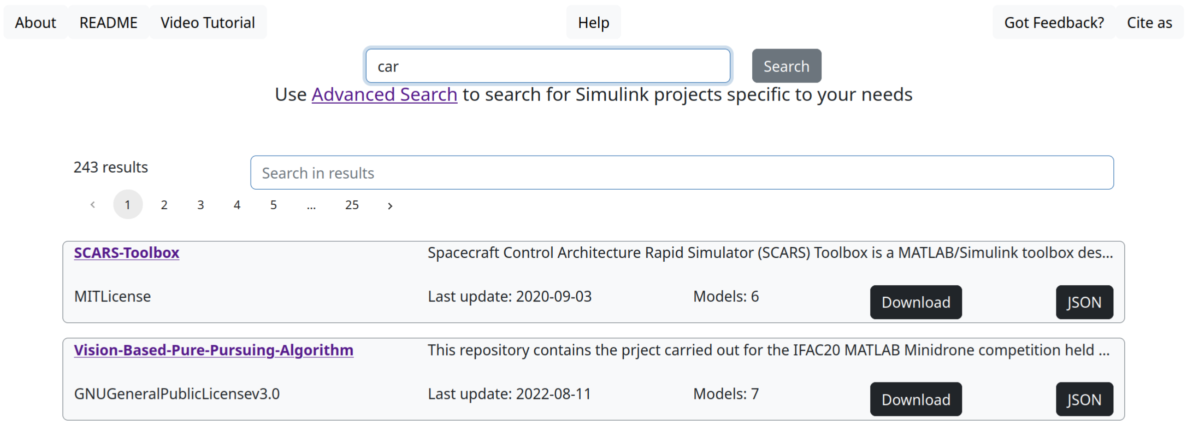
While the GitHub API exposes a project's license name, MATLAB Central projects are bundled with license files that require further analysis. To automate this process, we utilized an open-source library employed by GitHub [347]. Mining MATLAB Central took about two and a half days and yielded 4.2k projects with 18 attributes [354].

Model Metrics

To facilitate searching based on Simulink model metrics, we extended the existing SLNET-Metrics tool to add more metrics, including the presence of a TargetLink blocks, toolbox dependencies, and system target files. Responding to survey responses (which highlighted researchers' interest in filtering models based on block categories), we further enhanced the tool to support the categorization of block types into non-overlapping categories, as employed in our recent work [10]. We analyzed models on MATLAB R2022b and collected 39 model metrics [354] overall.

2.2.6 User Interface

ScoutSL has simple and advanced search. The latter offers three distinct user interfaces: Simulink model search, repository search, and commit search (catering to various dataset research requirements, including model evolution studies and subject models for tool evaluations).



Use Advanced Search to search for Simulink projects specific to your needs

243 results

SCARS-Toolbox	Spacecraft Control Architecture Rapid Simulator (SCARS) Toolbox is a MATLAB/Simulink toolbox des...			
MITLicense	Last update: 2020-09-03	Models: 6	Download	JSON
Vision-Based-Pure-Pursuing-Algorithm	This repository contains the prject carried out for the IFAC20 MATLAB Minidrone competition held ...			
GNUGeneralPublicLicensev3.0	Last update: 2022-08-11	Models: 7	Download	JSON

Figure 2.17. Example simple Simulink project search for “car”.

Simple Search

In the simple search users enter a text-based query, which ScoutSL matches with the project descriptions in the database. An example search of “turbine” produces 50+ project results. ScoutSL then sorts the results in descending order based on a ranking score. To compute the ranking score, we adapted a strategy inspired by previous work [224] that aimed to classify engineered and toy projects. We selected the common Table 2.5 attributes shared by GitHub and MATLAB Central projects and scored them based on survey responses. To prioritize Git-based attributes highlighted by the survey, we assigned lower weight scores to model revision and model contributors, while MATLAB Central projects received zero for these attributes.

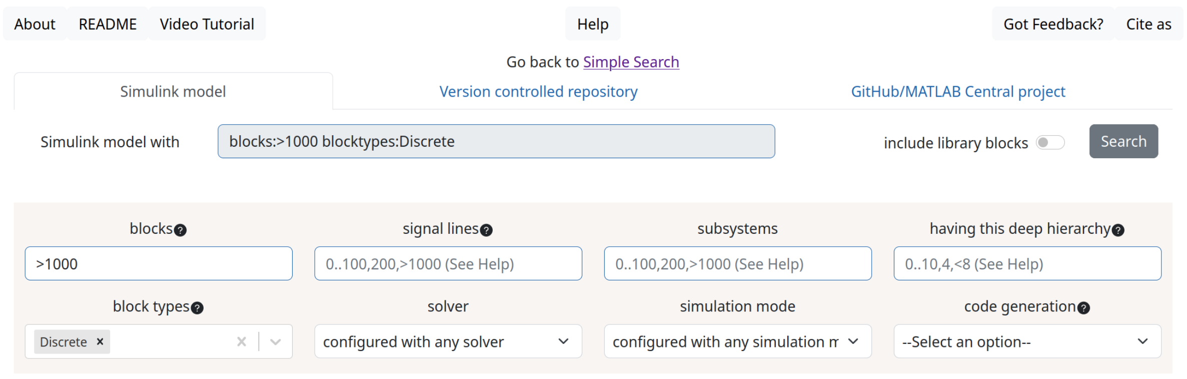
We scored each attribute either via a binary (0 or 1) or a continuous scheme. For the latter we considered the distribution of data attributes, filtered out outliers, and normalized the scores to between 0 and 1. The final project score was calculated by summing the weighted Table 2.5 scores. While the scoring scheme is based on our survey responses, we do

Table 2.5. Project scoring scheme; CC = cyclomatic complexity.

Attribute	Survey	Weight	Scoring Scheme
Blocks	Size (12)	15	Continuous
Block types	Property (11)	15	Continuous
Code generation	Property (11)	15	Discrete (0,1)
Test harness	Property (11)	15	Discrete (0,1)
Documentation	Property (11)	15	Discrete (0,1)
License	License type (7)	10	Discrete (0,1)
CC	Complexity (5)	5	Continuous
Model revision	Git (2)	2	Continuous
Model contributors	Git (2)	2	Continuous
Toolbox	Add-ons (1)	1	Discrete (0,1)

not make claims regarding the projects' engineering quality [224]. Evaluating and improving the scoring scheme is future work.

Advanced Search: Simulink Model



Simulink model with: blocks:>1000 blocktypes:Discrete

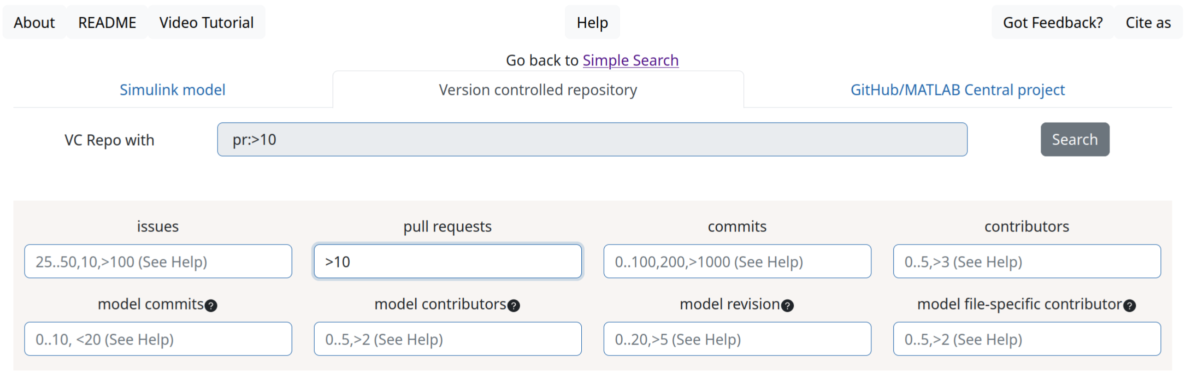
blocks: >1000
signal lines: 0..100,200,>1000 (See Help)
subsystems: 0..100,200,>1000 (See Help)
having this deep hierarchy: 0..10,4,<8 (See Help)
block types: Discrete
solver: configured with any solver
simulation mode: configured with any simulation m
code generation: --Select an option--

Figure 2.18. Example search for Simulink models that contain over 1k blocks, including some discrete blocks.

To cater to the goal of facilitating Simulink model sampling, Figure 2.18 illustrates ScoutSL's model search UI. The page highlights the specific model metrics that users expressed interest in, as determined through our survey. Users can input numeric values or select attribute options from drop-down menus to refine their search criteria. For example, they can search for Simulink models with over 1k blocks having discrete blocks. This search query generates a list of over 650 projects as search results.

Advanced Search: Simulink GitHub Repository

In order to support studies on model evolution and changes, ScoutSL incorporates GitHub-based selection criteria focused on version control and project management, as seen in Figure 2.19. Specifically, users can employ search criteria such as the number of project issues, pull requests, commits, and contributors. With our emphasis on Simulink models,



About README Video Tutorial Help Got Feedback? Cite as

Go back to Simple Search

Simulink model | Version controlled repository | GitHub/MATLAB Central project

VC Repo with: pr:>10 [Search]

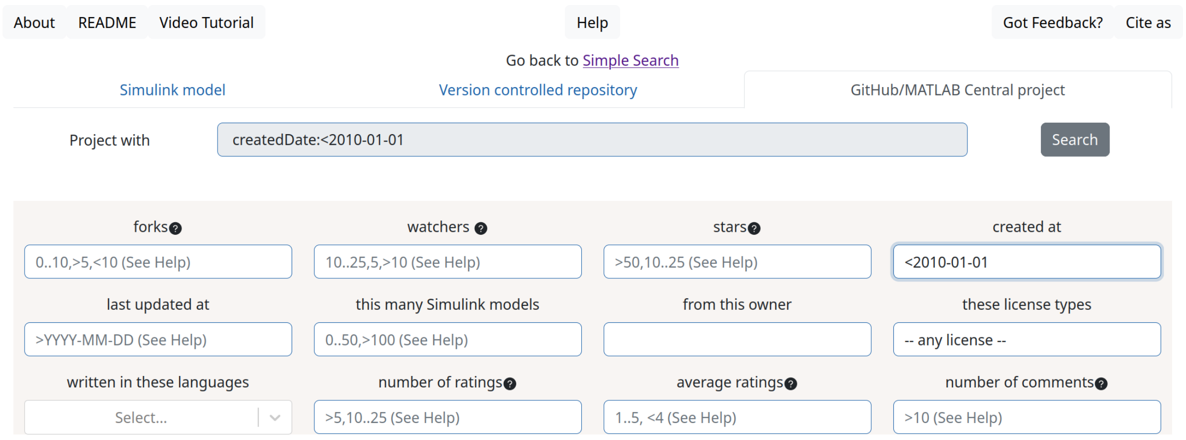
issues	pull requests	commits	contributors
25..50,10,>100 (See Help)	>10	0..100,200,>1000 (See Help)	0..5,>3 (See Help)
model commits	**model contributors**	**model revision**	**model file-specific contributor**
0..10, <20 (See Help)	0..5,>2 (See Help)	0..20,>5 (See Help)	0..5,>2 (See Help)

Figure 2.19. Example search for Simulink GitHub repositories that have over 10 pull requests.

users can also search for model-specific commits and contributions, which are subsets of project commits and contributors.

Additionally, ScoutSL enables users to filter projects based on a specific number of model revisions or model files with a certain number of authors. These model-specific search criteria are a unique ScoutSL feature not available on other online web-based tool. While other tools may offer project-level commit information and allow to search for commits per project [303], ScoutSL offers to search over the entire project database. As an example, researchers investigating model development can efficiently identify relevant projects by using a search query that filters for those with a significant number of model revisions. By using a search query for projects with more than 10 model revisions, ScoutSL yields over 500 relevant projects.

Advanced Search: Simulink Project



About README Video Tutorial Help Got Feedback? Cite as

Go back to Simple Search

Simulink model | Version controlled repository | GitHub/MATLAB Central project

Project with: createdDate:<2010-01-01 [Search]

forks	watchers	stars	created at
0..10,>5,<10 (See Help)	10..25,5,>10 (See Help)	>50,10..25 (See Help)	<2010-01-01
last updated at	**this many Simulink models**	**from this owner**	**these license types**
>YYYY-MM-DD (See Help)	0..50,>100 (See Help)		-- any license --
written in these languages	**number of ratings**	**average ratings**	**number of comments**
Select...	>5,10..25 (See Help)	1..5, <4 (See Help)	>10 (See Help)

Figure 2.20. Example search for pre-2010 Simulink projects.

A commonly employed strategy in mining software repositories for exploratory studies is to sample projects based on popularity metrics. ScoutSL provides the capability to filter projects based on such metrics, as depicted in Figure 12. Additionally, users are able to query projects created within specific date ranges. Another feature we offer is the ability to filter projects based on license type, as it was identified as a requested feature in the survey responses. Since a Simulink project is often accompanied by complementary scripts written in other programming languages, users can also perform searches involving such criteria. For instance, researchers interested in studying projects Simulink projects with JAVA and C code can use ScoutSL to get 250+ relevant projects. While the most up-to-date project attributes may be available through the primary hosting sites, our database exclusively comprises Simulink projects, which are not easily sampled using existing tools or the primary hosting sites from which we mine our projects.

2.2.7 Related Work

While several tools have been developed to facilitate sampling of open-source projects from platforms like GitHub, they primarily focus on textual programming languages [313, 321]. To obtain model artifacts, a web-based search tool, ModelMine [303], queries GitHub API to narrow down results with file extension. However, the tool mistakenly identifies ".simulink" as a Simulink model file extension and is inherently limited by GitHub API's 1k results per request limit.

A recent study examining forums of modeling tools including MATLAB/Simulink highlighted the potential benefits of model repositories, particularly for novice users who may encounter difficulties when attempting to model something specific [340]. The study emphasizes the importance of establishing and maintaining a diverse repository of example models. To that end, MAR [334] is a web-based search engine that maintains metamodels for various types of models, including UML models. For Simulink, the tool analyzes the pre-curated corpus to extract their metamodel using a third-party tool. As such, using MAR requires knowledge of modelling languages like EMF to get relevant models, and the search space is limited to 200 Simulink models.

2.2.8 Conclusions and Future Work

ScoutSL (`http://scoutsl.net`) is the first search engine geared towards Simulink users' needs. ScoutSL allows searching over 18k Simulink projects containing over 100k Simulink models.

Future works include extension of mining tool to enlarge and augment the dataset with new primary as well as derived project/model attributes such as project domain, Simulink model version. We intend to improve the search engine performance and conduct a thorough evaluation.

Acknowledgements

Christoph Csallner has a potential research conflict of interest due to a financial interest with Microsoft and The Trade Desk. A management plan has been created to preserve objectivity in research in accordance with UTA policy. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1911017 and a gift from MathWorks.

3 Empirical Research with Open-Source Simulink Models

This chapter demonstrates that open-source Simulink models, despite their limitations, are suitable for empirical research through analysis and case studies. The chapter consists of the publications *Characteristics, potentials, and limitations of open-source Simulink projects for empirical research* (Section 3.1), *RaQuN: a generic and scalable n-way model matching algorithm* (Section 3.2), *Simulink bus usage in practice: an empirical study* (Section 3.3), and *Beyond code: Is there a difference between comments in visual and textual languages?* (Section 3.4).

Theoretical Assessment of Dataset Suitability

Study: Empirical Potential of Open-Source Models The Simulink model scarcity that we diagnosed in Chapter 2 has one obvious mitigation: to find and collect open-source models, and make them permanently publicly accessible in a dataset. Over the years, a number of ever-larger, and more sophisticated Simulink model datasets were created by Chowdhury *et al.* [235, 236] and Shrestha *et al.* [337, 352, 10]. Our work *Characteristics, potentials, and limitations of open-source Simulink projects for empirical research* falls chronologically between these studies. In this work, we analyze the (at the time) largest Simulink model dataset [236] for its suitability in empirical research. We make the following contributions: (i) we determine various characteristics of the models and projects from the dataset, (ii) we critically evaluate the datasets' theoretical suitability for empirical research along multiple dimensions, and (iii) provide a stable snapshot of the models and their projects facilitating comparative studies on a stable dataset. We found the dataset to be highly diverse in terms of most metrics we studied. This diversity ranges from small, basic examples up to sophisticated projects with lifespans of multiple years.

Context and Subsequent Work However, very few open-source models and projects reach the quality or scale of industry models, which motivates our work in Chapter 4. We conclude that the open-source models are suitable for empirical research, provided researchers filter the model dataset according to their research needs.

Since our study was published, the much larger dataset SLNET [337] was created by Shrestha *et al.*, encompassing ca. 9,000 models whose licenses allow for redistribution. The latest dataset we know of is EvoSL [10] which was specifically mined from Simulink-based GitHub projects to foster evolutionary studies. As the model datasets became larger, they also became unwieldy. This was the motivation for ScoutSL, a Simulink model search engine in Section 2.2.

Practical Assessment of Dataset Suitability with Case Studies

In addition to the theoretical insights from Section 3.1, we practically demonstrate the suitability of open-source models and their datasets by employing them in our studies. In fact, each of our works in Sections 3.2 to 3.4 and later in Sections 4.1 and 4.2 can be seen as a case study demonstrating their suitability, as each of our case studies would not have been possible without suitable open-source models.

Study: Model Matcher RaQuN To evaluate the model matching tool RaQuN in *RaQuN: a generic and scalable n-way model matching algorithm* on Simulink models, we extracted and collected model variants, *i.e.*, closely related, but not identical models. The model variants originate from prior case studies and us mining them from GitHub projects. The Simulink models enriched the prior evaluation [323] with a completely new type of models that was among the most challenging in terms of quality and runtime for the evaluated matching tools.

Study: Simulink Buses In *Simulink bus usage in practice: an empirical study*, we do not evaluate a research prototype but study Simulink models and their development process, particularly for the practical use of specific model elements: buses, *i.e.*, a signal grouping mechanism. We mined Simulink models from GitHub and found buses in 433 of 4,812 models, giving us a large enough sample size for various fruitful statistical analyses. All hypotheses derived from our statistical analysis were confirmed with by practitioners in a confirmatory survey, showing the value of deriving usage patterns from open-source models.

Study: Simulink Comments Datasets like SLNET [337] often collect complete Simulink projects and not only the Simulink model files. We leveraged this fact in our study *Beyond code: Is there a difference between comments in visual and textual languages?* to study commenting practices not only in Simulink models, but also their accompanying MATLAB source code. While Simulink projects are different from projects written in textual programming languages (Java, Python, Smalltalk) along various dimensions, the similarities in commenting practices were surprising.

Other Case Studies Both tool studies in Chapter 4 can be viewed as further case studies showing the versatility of SLNET, as they present many challenging edge cases in their evaluation. Particularly the diversity of the models is a key ingredient for GRANDSLAM's fuzzing aspect. When GRANDSLAM generates models by mining SLNET's outdated, rarely used, or incomplete models, the likelihood of exposing bugs in the Simulink toolchain increases significantly.

In addition to our work, contemporary studies regularly employ open-source models, *e.g.*, the SLNET dataset alone was used at least in seven publications [10, 5, 356, 11, 6, 372, 373] and our submitted works [7, 8]. Lastly, we are currently working on a project that does not use SLNET's Simulink models but its MATLAB files only. In *Can We Make Code Green? Understanding Trade-Offs in LLMs vs. Human Code Optimizations* [371], we investigate the energy efficiency of MATLAB code and try to optimize its energy efficiency with various methods.

Summary and Context Together, this chapter's results provide evidence that open-source models – when selected carefully – enable rigorous research, from tool evaluation to evolutionary studies. This chapter lays the groundwork for Chapter 4, where we address remaining gaps in scale and realism.

3.1 Characteristics, potentials, and limitations of open-source Simulink projects for empirical research

↪ Thesis Roadmap

Authors: Alexander Boll, Florian Brokhausen, Tiago Amorim, Timo Kehrer, and Andreas Vogelsang.

Published in: Software and Systems Modeling, 2021.

Abstract: Simulink is an example of a successful application of the paradigm of model-based development into industrial practice. Numerous companies create and maintain Simulink projects for modeling software-intensive embedded systems, aiming at early validation and automated code generation. However, Simulink projects are not as easily available as code-based ones, which profit from large publicly accessible open-source repositories, thus curbing empirical research. In this paper, we investigate a set of 1734 freely available Simulink models from 194 projects and analyze their suitability for empirical research. We analyze the projects considering (1) their development context, (2) their complexity in terms of size and organization within projects, and (3) their evolution over time. Our results show that there are both limitations and potentials for empirical research. On the one hand, some application domains dominate the development context, and there is a large number of models that can be considered toy examples of limited practical relevance. These often stem from an academic context, consist of only a few Simulink blocks, and are no longer (or have never been) under active development or maintenance. On the other hand, we found that a subset of the analyzed models is of considerable size and complexity. There are models comprising several thousands of blocks, some of them highly modularized by hierarchically organized Simulink subsystems. Likewise, some of the models expose an active maintenance span of several years, which indicates that they are used as primary development artifacts throughout a project's lifecycle. According to a discussion of our results with a domain expert, many models can be considered mature enough for quality analysis purposes, and they expose characteristics that can be considered representative for industry-scale models. Thus, we are confident that a subset of the models is suitable for empirical research. More

generally, using a publicly available model corpus or a dedicated subset enables researchers to reproduce findings, publish subsequent studies, and use them for validation purposes. We publish our dataset for the sake of reproducing our results and fostering future empirical research.

3.1.1 Introduction

Domain-specific models are the primary artifacts of model-based development of software-intensive systems [139, 211]. They serve as a central means for abstraction, facilitate analysis and simulation in the early stages of development, and provide a starting point for automated software production. Over the last two decades, Matlab/Simulink[1] (in the sequel referred to as Simulink, for short) has emerged in various domains (*e.g.*, automotive, avionics, industrial automation, medicine) as a de facto standard for the industrial model-based development of embedded systems [83].

However, Simulink projects and models created and maintained in an industrial context are usually not publicly available due to confidentiality agreements or license restrictions [124, 146, 228, 283]. Access to these models, in general, is limited, making research results hard if not impossible to reproduce [9]. Publicly available projects do not reflect "real world models" [208, 218, 234, 267], which severely limits empirical research. Additionally, there are no commonly established benchmarks for assessing and comparing the effectiveness of new techniques and tools, and little is known about the usage of these models in practice. As a consequence, scientific insights into model-based development with Simulink are not nearly as deep and substantial as for classical code-based development, which highly profits from large publicly available open-source software repositories [65, 74, 75, 169, 185].

As a step to overcome this situation, we investigate a set of 1,734 freely available Simulink models from 194 projects, originally collected by Chowdhury et al. [236] and updated in terms of our study. The set comprises projects from Matlab Central[2], SourceForge[3], Github[4], and other web pages, as well as two smaller sets [148, 290]. We first analyze these projects and models concerning their basic characteristics, including (i) their development context, (ii) their complexity in terms of size and model organization within projects, and (iii) their evolution over time. Thereupon, we discuss the corpus' potentials and limitations for empirical research.

We found that the projects and models comprised by the corpus are very heterogeneous concerning these characteristics. For (i), the projects stem from different origins and application domains. Most projects come from academia, and the distribution over application domains is skewed towards the energy sector. For (ii) and (iii), most of the projects are relatively small, exposing a short lifetime and hardly any collaborative development effort. Many of them are toy examples with limited practical relevance. However, some large-scale projects provide sophisticated Simulink models in a mature project structure, and the most

[1] `http://www.mathworks.com/products/simulink` (visited on 15/12/2025)
[2] `https://www.mathworks.com/matlabcentral/fileexchange` (visited on 15/12/2025)
[3] `https://sourceforge.net` (visited on 15/12/2025)
[4] `https://github.com` (visited on 15/12/2025)

long-living projects have a lifetime of several years of active development. Besides these limitations, our results show that there are also potentials for empirical research in the Simulink area. According to our results' validation with a domain expert, many models expose several characteristics that can be considered representative of industry models, and are suitable for empirical research, the circumstances of which are discussed in this paper. The validity of this study may be threatened internally by a subjective classification of a project's context and externally by the limited size of our data set.

We publish the updated corpus[5] for the sake of reproducing our results and fostering future empirical research, which is the major impact we aim for with this paper.

3.1.2 Model-based Development with Simulink

Simulink is a Matlab-based graphical programming environment for modeling, simulating, and analyzing multi-domain dynamical systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. Different kinds of blocks can be connected via ports to transmit outputs and receive inputs, thus yielding a dataflow-oriented model. Subsystems are special blocks that contain another Simulink diagram, thus enabling hierarchical modeling.

Figure 3.1 shows an example of a Simulink diagram (taken from [222]). The model shows a dual-clutch control of an automatic transmission system of a vehicle with two separate clutches. Blocks of various types are connected via signal lines. The four smaller blocks on the left side are inport blocks, which transport input values from the model's context. One of them is the car's current speed ($VehSpd$), which is further processed to compute the next gear shift. Also, there are three outport blocks (same symbol as inports but with incoming signal lines), which transport output values of the model to its context. The four rectangular blocks shaded in gray are subsystems. The subsystems are part of the model, and the contained behavior can be displayed on request. The other shapes represent basic blocks (*i.e.*, non-composite blocks). The pentagon at the top (*trq_dem*) is a *goto* block that transports its signal to some other part of the model (to a point deeper in one of the subsystems). The triangle ($Tmax$) is a *gain* block, which multiplies a signal with a constant. The black bar is a multiplexer block, which combines inputs with the same data type and complexity into a vector output. The rectangle with the label "[0,1]" is a saturation block, which produces an output signal that is the input signal's value bounded to some upper and lower values.

The process of computing the states of a Simulink model at successive time steps is known as *solving* the model. This way, models can be simulated for the sake of validation or verification. Simulink comes with two classes of solvers: (1) Fixed-step solvers, as the name suggests, solve the model using the same step size from the beginning to the end of the simulation; (2) Variable-step solvers vary the step size during the simulation. Within one simulation step, each block of the model updates its state and output values according to the specified behavior and this step's input values.

Several add-on tools allow to model state-based systems (*Stateflow*), generate code from Simulink models (*e.g.*, *TargetLink*, *Embedded Coder*), or to do formal verification and test case generation (*e.g.*, *Design Verifier*).

[5] https://doi.org/10.6084/m9.figshare.13636589

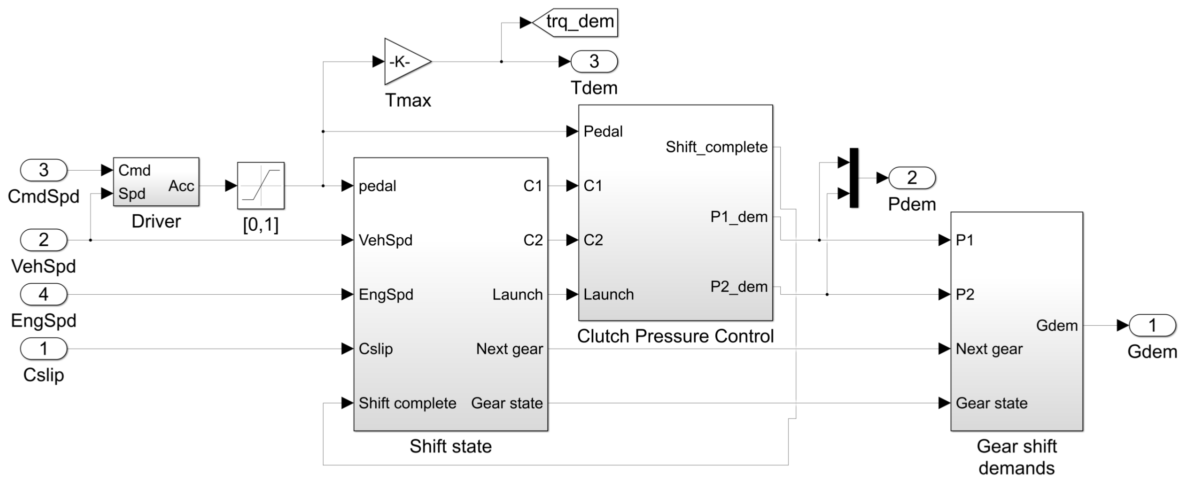


Figure 3.1. Example of a Simulink block diagram modeling the dual-clutch control of an automatic transmission system of a vehicle with two separate clutches [222].

3.1.3 Related Work

Empirical Studies on Model Characteristics

Several authors have investigated existing UML models regarding their characteristics and perception. Hebig et al. [189] released the currently largest set of open source UML models mined from GitHub repositories (the *Lindholmen* dataset). They described content- and process-related characteristics of the UML models and the corresponding projects. In a follow-up study [227], they triangulated their results with qualitative surveys. They found that collaboration seems to be the most important motivation for using UML, and teams use UML during communication and planning of joint implementation efforts. Störrle analyzed the impact of UML diagram size on the understanding of the diagrams [255]. He found a strong negative correlation between diagram size and modeler performance. He used in his experiments class diagrams, state charts, and sequence diagrams with a mean diagram size of 25 to 30 elements, which were smaller than the models found in the Lindholmen dataset [189].

Regarding works describing characteristics of Simulink models, Dajsuren et al. [127, 167] reported coupling and cohesion metrics they found in ten industrial Simulink models. They measured the inter-relation of subsystems as well as the inter-relation of blocks within subsystems. Stephan et al. [157] developed a taxonomy to describe Simulink model mutations. The mutations are organized by categories based on the types of model clones (Type 1, 2 or 3) they inject, and further broken down into mutation classes that resemble typical edit operations on Simulink models. In order to evaluate the representativeness of the edit operations, the taxonomy has been applied to three Simulink projects, two of them being publicly available. Although the work's main aim was to establish a framework for evaluating model clone detectors, the taxonomy can be considered general enough to describe aspects of Simulink model evolution from a qualitative perspective. Kehrer *et al.* [102] defined Simulink model editing operations which have been used to study the evolution of a cruise control model. Balasubramaniam *et al.* [288] conducted an empirical study investigating the types and quantity of software changes in the context of embedded control systems. Insights are gained from two widely adopted open-source control software suites, namely ArduPilot and Paparazzi UAV. These are used to develop a code mutation framework mimicking typical evolution in control software. Later on, they apply this framework to explore the impact of software evolution on the behavior of three controllers designed with Simulink, focusing on the mismatches that arise between control models and the corresponding control software. Chowdhury et al. [236] reported on a large set of freely available Simulink models that they crawled from various sources on the Internet. They analyzed these models in terms of content and reported basic measures such as the number of blocks and connections. Their set is used in this paper as the basis for further analysis. Moreover, we update their corpus by collecting the latest project and model snapshots, and we extract additional meta-data from those projects hosted on GitHub to assess their evolutionary characteristics.

Relevance of Open Source Models for Empirical Research

Conducting extensive empirical studies in modeling and model-based development can be challenging due to the lack of repositories with large numbers of freely accessible models. Badreddin et al. studied 20 free open source software (FOSS) projects with high numbers of commits without finding UML and concluded that it is barely used in FOSS [124]. Similarly, Ding et al. found only 19 projects with UML when manually studying 2,000 FOSS projects from popular FOSS repositories [146].

Most empirical studies on modeling in practice are case studies analyzing limited sets of models in specific contexts (*e.g.*, [100, 113, 144, 239, 309]) or qualitative studies including interviews and surveys (*e.g.*, [133, 155, 259]). Some studies approach the use of models in FOSS from a quantitative perspective, studying a large number and variety of projects. For example, to study the use of sketches, Chung et al. collected insights from 230 persons contributing to 40 FOSS projects [89]. Langer et al. studied the lifespan of 121 enterprise architect models in FOSS projects [151]. Collections of models used for experimental evaluations of model-based development tools can be found, *e.g.*, in [81, 315] .

Some authors created datasets of Simulink models as benchmarks. For example, Bourbouh et al. [209] compiled a set of 77 Stateflow models to demonstrate the effectiveness of their tool. Similarly, Sanchez et al. [282] downloaded 70 Simulink models larger than 1MB from GitHub. Another benchmark of Simulink models was created as part of the Applied Verification for Continuous and Hybrid Systems (ARCH) workshop [262]. This benchmark offers six models in four projects used as study objects in a competition to solve a range of general problems (*e.g.*, falsification or model checking).

Due to the lack of publicly available models, most experimental evaluations of tools rely on models that have been synthetically created using a dedicated model generator [85, 104, 252, 300, 301, 306]. Specifically, there is a line of research on the generation of realistic models conducted by Yazdi *et al.* [142, 163, 183, 204]. The basic idea is to analyze model histories to learn statistical properties of model editing sequences, which are then used to configure a model generator that aims to generate realistic models by simulating such editing sequences.

Stol and Ali Babar performed a systematic literature review on empirical studies in FOSS [87]. Based on their observation of the analyzed studies' low methodological quality, they proposed a guideline for conducting empirical studies in FOSS projects. In the guideline, they emphasized the importance of reporting the characteristics of the analyzed sample: "Such details can include: the size of the FOSS software (expressed as lines of code), size of community (expressed as the number of active and inactive participants), and the domain of the FOSS software (*e.g.*, operating systems, desktop software, infrastructural such as web servers)." In our paper, we augment the meta-data of models with information about content, project context, and process context. This information allows researchers to derive and justify suitable samples for their research.

In code-based development, a large research community focuses on analyzing [65, 74, 75, 169, 185], and building [94] FOSS repositories, especially in the context of platforms with social features (*e.g.*, GitHub) [149, 213]. However, models known from model-based development, including UML, Simulink and other domain-specific kinds of models, have not yet made it into typical research on mining software repositories.

3.1.4 Study Design

Research Objective

Our research aims at *understanding the characteristics of publicly available Simulink models to assess their potential for empirical research.* We characterize these models according to three perspectives: *context*, *size*, and *evolution*. We selected these perspectives because we found indications in the literature that those perspectives should be considered when conducting empirical studies. For example, the ACM SIGSOFT Empirical Standards [302] lists *"describes the context of the case in rich detail"* as an *essential* part of any case study or action research study. Baltes and Ralph [289] argue that representativeness of corpora can be improved by "(1) Including artifacts from diverse domains (*e.g.*, aerospace, finance, personal computing, robotics). [. . .] (3) Making the corpus large enough to support heterogeneity sampling and bootstrapping. (4) Attempting to match the parameters we can discern [. . .]". Moreover, we considered existing studies on model characteristics (see Section 3.1.3) and found that these consistently report the context of the analyzed models, size and complexity [127, 255], and evolution [189].

Therefore, we analyzed the mentioned three perspectives as formulated by the following research questions.

RQ1: *In which context are Simulink projects created?*

Information regarding the project context is a necessary prerequisite to assess the external validity of any future empirical research based on our Simulink project corpus. For example, the validity of research results might be limited to dedicated application domains.

RQ2: *What is the size of the Simulink models and how are they organized within their defining projects?*

The primary motivation for assessing the size of models is that future benchmarks or experiments being based on our corpus might require models that exceed a certain degree of complexity and are not just toy models. In particular, we are interested in whether our corpus comprises industry-scale models useful for further research.

RQ3: *How do Simulink projects and their models evolve over time?*

The motivation to understand how Simulink projects and models evolve is to assess their

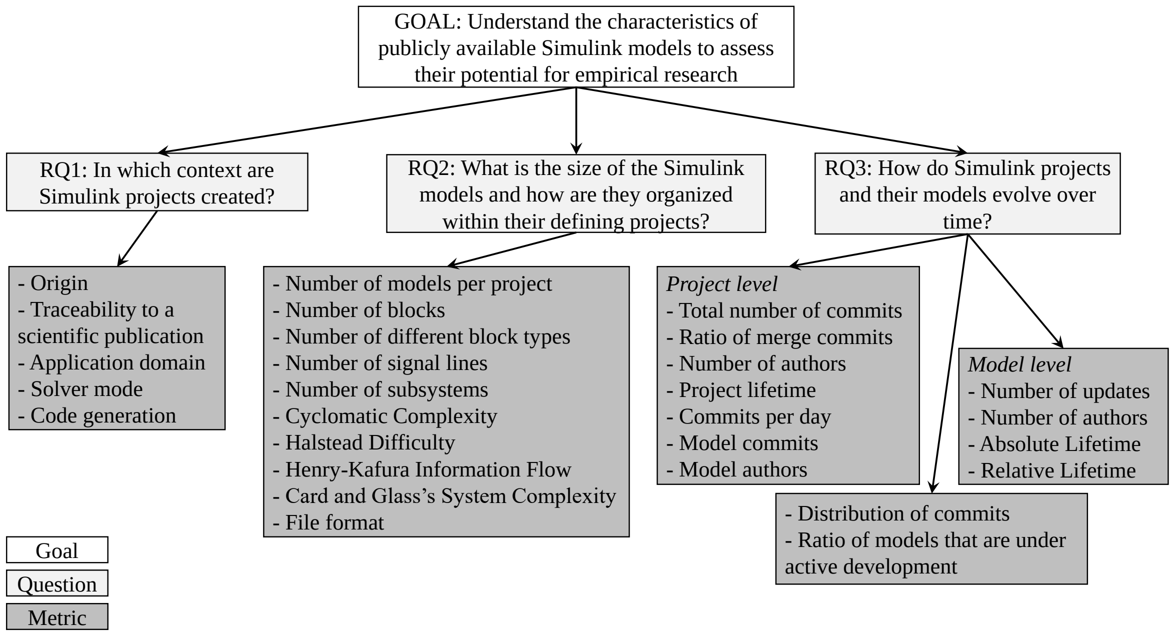

GOAL: Understand the characteristics of publicly available Simulink models to assess their potential for empirical research
RQ1: In which context are Simulink projects created?
RQ2: What is the size of the Simulink models and how are they organized within their defining projects?
RQ3: How do Simulink projects and their models evolve over time?
- Origin
- Traceability to a scientific publication
- Application domain
- Solver mode
- Code generation
- Number of models per project
- Number of blocks
- Number of different block types
- Number of signal lines
- Number of subsystems
- Cyclomatic Complexity
- Halstead Difficulty
- Henry-Kafura Information Flow
- Card and Glass's System Complexity
- File format
Project level
- Total number of commits
- Ratio of merge commits
- Number of authors
- Project lifetime
- Commits per day
- Model commits
- Model authors
Model level
- Number of updates
- Number of authors
- Absolute Lifetime
- Relative Lifetime
- Distribution of commits
- Ratio of models that are under active development
Goal
Question
Metric

Figure 3.2. Goal Question Metric model of the research objective

suitability for learning from their development history. Thus, we are particularly interested in whether there are any projects under active development or maintenance for a long time.

The research objective is summarized using a *Goal Question Metric* [46] model, illustrated in Figure 3.2. Metrics and respective extraction methods will be presented in detail in the remainder of this section.

Study Subjects

This study can be classified as a *quantitative and qualitative non-probability sample study* [254]. Our sample is based on the largest [291] set of publicly available Simulink models,[6] collected by Chowdhury et al. [236]. The set by Chowdhury *et al.* comprises a smaller Simulink model collection [148], a Stateflow model collection by the CoCo-Sim-Team [290], and many other projects from Matlab Central, Sourceforge, GitHub, and other sources such as web sites of university projects. Although critical open source repository sites could have been missed, the set by Chowdhury et al. covers a wide range of sources. Instead of using the provided dataset as it is, we re-collected a current snapshot in August 2020[7] consisting of all constituent Simulink models based on the information provided in the meta-data of the corpus of Chowdhury et al. The main motivation for this new snapshot arises from several inconsistencies we found between the actual corpus and the results presented in [236]. According to personal correspondence with the authors, these inconsistencies may originate from only a subset of the entire corpus models being used in their study. For many projects, the newer snapshot also provided updated models and a richer model evolution history for answering ***RQ3***.

[6] https://github.com/verivital/slsf_randgen (visited on 15/12/2025)
[7] https://doi.org/10.6084/m9.figshare.13636589

We collected the Simulink projects and models of our updated corpus using the project URLs provided in the original corpus' meta-data. Out of the 205 projects listed in the meta-data, 204 mention a URL. We visited these 204 web pages and found 193 to be still online. Out of the remaining 11 projects, we were able to find one additional project in the original dataset. In sum, we could thus analyze 194 projects comprising a total of 1,736 Simulink models. Two of these models are invalid and cannot be opened by Matlab Simulink, which reduces the number of actually analyzable models to 1,734.

Our corpus comprising all the projects and models used in this study, including the respective meta-data, is available in our reproduction package.

Data Analysis

RQ1: Project Context To get a basic understanding of the context in which a Simulink project has been created, we classify each project with respect to the following dimensions:

Origin: We use the categories *academia*, *industry*, and *Mathworks* to classify the origin of a project. The categorization is determined based on the affiliations of the developers associated with the projects. Although Mathworks' modeling projects might also be classified as industry projects, we assume that Mathworks developers do not represent typical "end-users" of Simulink from industry, which is why we differentiate among the two.

Application domain: We use the domains *energy*, *electronics*, *automotive*, *avionics*, *robotics*, *domain independent*, and *other* to classify the projects with respect to their application domain. Domain-independent projects commonly encompass tools (*e.g.*, Simulink analyzers, diagram layout managers or general toolboxes). These projects are assumed to be applicable in any domain.

Both classifications were performed manually by four researchers based on multiple sources of information. Besides the project data itself and the project's web site, a web search was performed to gather additional information like developer affiliation or associated scientific publications. Concerning the list of possible application domains, we first used the classification scheme of [310], which comprises eight application domains. We revised this initial classification scheme during our analysis, as some of the analyzed projects did not fit into that scheme and some domains were not represented by a single project. We ended up reusing their domains automotive, avionics, and "unclear". Newly created domains were decided upon together and were added if a project would not fit into another domain: energy, electronics, domain-independent, robotics, military, biology, wearable. The domains railway and automation were adjusted to transport and home automation. The domains finance, health care, public, and telecommunication were not used in our classification. To mitigate classification errors, a consensus needed to be reached among the four researchers, which was achieved through iterative discussions. Without a consensus, no useful project description available, or when a project domain remains unclear, we used the categories *unclear* and *not enough information* for domain and origin, respectively.

Traceability to a scientific publication: Moreover, particularly for those projects originating from an academic context, we are interested in whether there is a scientific publication associated with the project. Such publications may be useful for additional research as they provide a more detailed context of a project and its models. It also gives an insight into the scientific diligence of the academic work.

Solver mode: The usage of a fixed-step solver might indicate that a model is used for code generation; otherwise, the model may only be used for simulation or other abstract purposes. If the model state is changed in fixed time steps, code generation for hardware on embedded systems as a deployment target is possible. The solver mode can be automatically extracted using the Simulink API.

Code generation using a standard code generator: In addition to the extraction of the solver mode, we searched for TargetLink, and Embedded Coder traces in the models, as these are the two most commonly used code generators in model-driven development with Simulink.

RQ2: Size and Organization of Simulink Models To characterize the size and complexity of the Simulink models and their organization within projects, we use a collection of standard Simulink model and complexity metrics from Olszewska et al. [196]. We collect and report measurements on a model level and on a project level by aggregating the measurements from a project's models. Our Matlab and Python scripts used for computation of all the metrics are published on Figshare.[8]

The *number of models per projects* helps to judge the projects' overall size. The *number of blocks* further helps in assessing the size of the models as well as, in an aggregated manner, the magnitude of the projects (*i.e.*, if projects consist of multiple small models or fewer large ones). We include masked subsystems in the calculation of the number of blocks. The *number of different block types* used in a model represents the modeling diversity. When aggregated, this metric serves to judge and compare the modeling diversity on a project level. Further, the comparison of model and project block diversity yields insights into how the different models within projects are modularized, *i.e.*, if the models of a project contain similar blocks or not. The *number of signal lines* represents the connectivity within the models, demonstrating the complexity of interaction between different functional blocks. This metric is also analyzed regarding the number of blocks to examine if there is a correlation. The *number of subsystems* characterizes a model from an architectural point of view and gives a hint on its modularization.

To assess the complexity of the models in our corpus, we use several complexity metrics that have been proposed by Olszewska et al. [196]. In their work, the authors have adapted several well-known code complexity metrics to Simulink. *Cyclomatic Complexity* was first introduced by McCabe [26] and assesses a program's complexity by counting the independent paths of program flow. Olszewska et al. adapt this for Simulink by mapping conditional statements of C to corresponding blocks in Simulink.

The Halstead metrics are another set of complexity metrics. For our study, we measure the Halstead Difficulty, which is calculated by the following formula:

$$D = \frac{n_1}{2} * \frac{N_2}{n_2}$$

where n_1 is the number of distinct Simulink block types, n_2 is the number of distinct input signals, and N_2 is the total number of input and output signals [196].

[8] https://doi.org/10.6084/m9.figshare.13636589

The *Henry-Kafura Information Flow* defines a subsystem's complexity based on the fan-in and fan-out of information flow for that subsystem. It is calculated as

$$HKIF = size * (fanIn * fanOut)^2$$

where *size* is the number of contained blocks (including subsystem blocks), $fanIn$ and $fanOut$ represent the number of afferent and efferent blocks of a subsystem. [196].

As last complexity metric, we calculate Card and Glass's *System Complexity*. System Complexity adds up two sub-metrics: Structural Complexity and Data Complexity ($SC = StructC + DataC$). Structural complexity is defined as the mean of squared values of fan-out for all subsystems (n):

$$StructC = \frac{\sum_{i=1}^{n} fanOut_i^2}{n}$$

Data Complexity is defined as a function that is directly dependent on the number of input and output signals (S_i) and inversely dependent on the number of fan-outs in the module:

$$DataC = \frac{1}{n} * \sum_{i=1}^{n} \frac{S_i}{fanOut_i + 1}$$

In addition to size and complexity, we assess the *file format* of the model. This information originates from the default file format of Simulink models changing from *.mdl* to *.slx*-extensions with the second annual release of Simulink in 2012. The file format may be necessary for tool and organization compatibility across versions. Finally, we establish any peculiarities to be observed between the manually identified domains of projects (*e.g.*, if industry projects differ from academic ones). Additionally, the identified industries themselves are analyzed concerning project and model structure.

RQ3: Project and Model Evolution We use meta-data extracted from a version control system to get an overview of how Simulink projects and models evolve, as it is customary in the field of software repository mining [74, 169]. We use a subset of our corpus' projects hosted on GitHub. We analyze only this subset as we can easily access the commit history of these Git projects[9]. On the contrary, the commit history is not provided for the other projects, thus excluded here. The GitHub subset comprises 35 projects containing 579 models, accounting for 18 % of all projects and 33 % of all models of the entire corpus.

On the project level, the *total number of commits* assesses the project's general development activity. In contrast, the *ratio of merge commits* to total commits and the *number of authors* serve as indicators of how the development is performed collaboratively. By extracting the *project lifetime*, we determine whether there are any long-living projects. We are particularly interested in whether any projects are actively maintained over time or whether they are just stored in the repository. Therefore, the lifetime comprises the time between the first and the last commit. Further, the *commits per day* provide a first indicator of how frequently the projects are updated. Since the total commits extracted from GitHub refer to all the project files, we are additionally interested in the proportion of *model commits*

[9] `https://git-scm.com` (visited on 15/12/2025)

that change at least one Simulink model in the project. Similarly, regarding the total number of project authors, we are also interested in the authors' proportion that changed at least one model file (*model authors*) during a project's lifetime.

On the model level, the *number of updates* reports the number of commits on the model file. The *number of authors* per model reports the number of different committers that have modified this model at least once. The *lifetime in days* encompasses the time between the first and the last commit that changed the model. In contrast, the *lifetime in %* denotes the model's relative lifetime concerning the overall lifetime of the project that comprises the model. We compute all models' lifetime according to the model files' inherent meta-data in addition to these metrics extracted from GitHub meta-data. This information provides a date for the first creation date and the last modification used to calculate the *lifetime in days* of all models.

Finally, we are interested in how the development workload, particularly the number of model modifications, is distributed over the lifetime of projects and models. Therefore, we calculate the *distribution of project commits*, the *distribution of model commits*, and the *ratio of models that are under active development* over the lifetime of a project. We assume a model to be under active development between the first and last commit of a model.

3.1.5 Study Results

In this section, we report the results of the analyses detailed in Section 3.1.4, structured by our research questions ***RQ1–RQ3***.

RQ1: Project Context

Origin and traceability to a scientific publication: As illustrated in Figure 3.3, of 194 projects, 113 (58 %) originate from an academic context, 34 (17 %) are provided by Mathworks, and 25 (13 %) projects are from industry. We could not classify the origin of the remaining 22 (12 %) projects due to missing information. We measured the agreement of the manual classification by computing Krippendorff's Alpha [103] to be 0.85, which is reliable (Alpha values $\geq$ 0.80 are considered reliable). Further, we found links to scientific publications for 26 (13 %) projects. Of these projects, 18 (9 %) originate from an academic context, while the remaining eight (4 %) projects are not classified concerning their context.

Application domains: Figure 3.4 shows the results of our domain classification. Most projects represent applications in the energy sector: 52 (27 %). The second-largest domain is electronics, with 47 (24 %) projects. These two domains make up more than half of all projects. 36 (19 %) of the models are classified as "domain-independent" – *e.g.*, used to demonstrate a Simulink tool. The avionics, robotics, and automotive domain together make up another 21 %.
The remaining categories are summarized as "other" in Figure 3.4. The "other" categories comprise 18 projects (9 %): six of which we could not classify at all and are thus "unclear", five telecommunication projects, and two audio projects. The remaining domains only contribute a single project to the corpus: biology, home automation, wearable, transportation, and military.
A Krippendorff Alpha of 0.86 was computed for the researchers' inter-rater agreement for

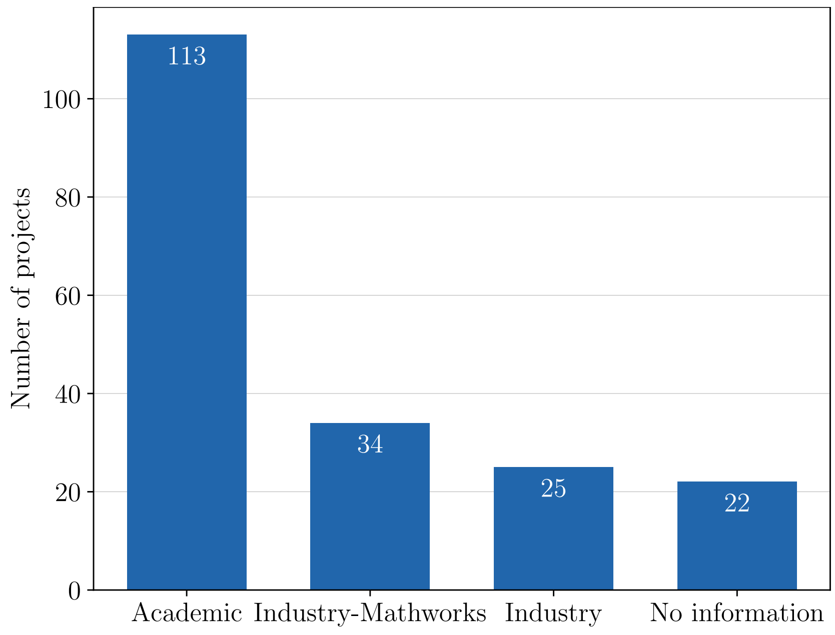


Figure 3.3. Origin of the projects comprised by our corpus.

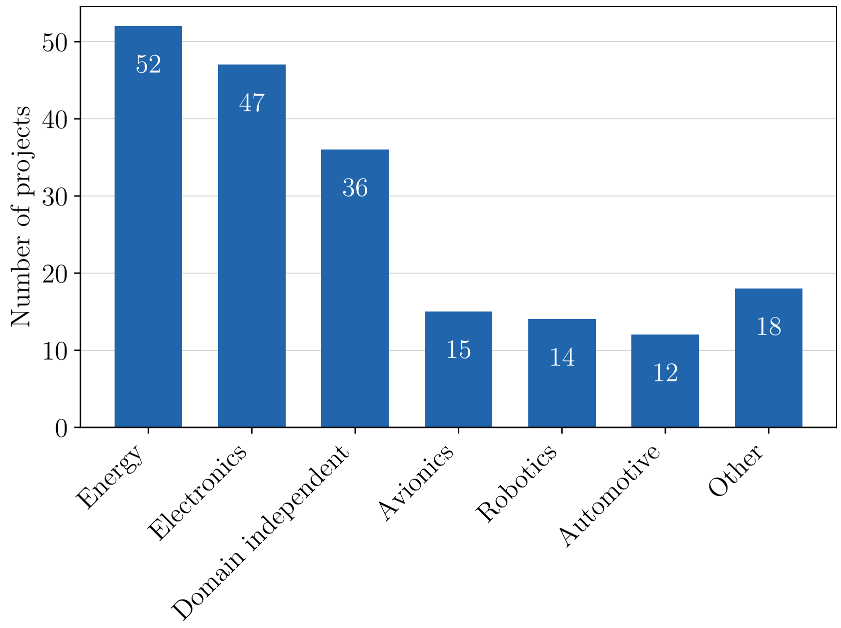


Figure 3.4. Application domains in which the projects have been developed.

domain classification, which, analogously to the original classification, can be considered reliable.

Solver mode and code generation: In our corpus, 576 models (33 %) use a fixed-step solver mode, while the remaining 1,158 (67 %) models apply a variable-step solver mode. Therefore, with two-thirds of the models, the majority of models use variable-step solvers.

When aggregating this to the project-level, there are 119 projects (61 %) that exclusively contain models applying a variable-step solver and 28 projects (14 %) use fixed-step solvers only. Another 47 projects (24 %) exhibit a mixture of both. Surprisingly, none of the models uses one of the two most widely established code generators, TargetLink,[10] and Embedded Coder.[11]

RQ1: *In which context are Simulink projects created?*
We were able to determine the origin of 172 projects, 65 % of those originate from an academic context and 17 % of projects are from industry. Further, 13 % of all projects are associated with a scientific publication. 27 % of all projects are developed in the energy domain, and another 24 % are from the electronics domain. Although one-third of all models use fixed-step solvers, none uses a standard code generator such as TargetLink or Embedded Coder.

RQ2: Model Sizes and Project Organization

Overall project comparison: Table 3.1 shows the different metrics on the project and model level. On the project level, all metrics are aggregated for all models in each project. The model-level metrics are calculated without relation to the projects.

The standard deviation is much larger for all metrics than the mean values (except for the number of different block types), which shows the diverse range of projects and models. Further, for these metrics, the median is significantly smaller than the mean value. Therefore, the metric values are rather small for most models and projects, while some exceptions are much larger than the median accounts. Some of the models are even empty, in the sense that they do not contain a single line or block. Models without any signal lines are usually library models, which are merely a collection of different types of blocks. In contrast, empty models might indicate “orphan models” or models that are not yet under active development.

The number of models per project varies substantially. Ninety-eight projects (50.5 %) contain only one model. The five largest projects contain 42 % of all models. Overall, no predominant trend is evident concerning the number of blocks, neither per model nor per project.

The number of signal lines also varies substantially between projects and models. Sixty-eight library models contain only blocks but no signal lines. Additionally, some model files contain just a single high-level subsystem, which is a reference to another model in the project. On the other hand, 450 models show high connectivity with equally as many or more signal lines than blocks. Apart from these two extremes, there is no common scheme in the correlation of signal lines and blocks. This phenomenon is also apparent when analyzing the average signal lines per block over all models, which amounts to 0.82 ± 1.52. One may expect models to contain more signal lines than blocks, in general. However, many models are library models, one signal line can connect more than two blocks, and models can contain descriptions or comment blocks that are not connected to other blocks.

[10] https://www.dspace.com/en/pub/home/products/sw/pcgs/targetlink.cfm (visited on 15/12/2025)
[11] https://www.mathworks.com/products/embedded-coder.html (visited on 15/12/2025)

Table 3.1. Calculated metrics for the project size. The metrics are reported per *project* and per *model*.

Metric	per	Min.	Max.	Mean	Median	Std. Dev.
Number of Models	project	1.0	252.0	8.93	1.00	25.84
	model	-	-	-	-	-
Number of Blocks	project	2.0	319,108.0	6,076.56	553.50	28,786.19
	model	0.0	59,860.0	679.85	70.00	2,746.76
Number of Block types	project	2.0	199.0	47.59	40.00	36.46
	model	0.0	66.0	16.11	13.00	12.97
Number of Signal lines	project	0.0	41,340.0	1,137.76	158.00	3,863.64
	model	0.0	12,844.0	127.29	28.00	470.32
Number of Subsystems	project	0.0	3,372.0	118.63	18.50	391.63
	model	0.0	1,791.0	13.27	4.00	50.14
Cyclomatic Complexity	project	1.0	384.0	14.45	2.41	37.49
	model	1.0	859.0	10.50	1.00	39.68
Halstead Difficulty	project	0.0	118.0	16.55	11.53	19.99
	model	0.0	118.0	10.37	0.00	15.99
Henry–Kafura Information Flow	project	0.0	7,936.0	161.49	1.06	828.75
	model	0.0	96,042.9	131.43	0.00	2,393.25
System Complexity	project	0.0	1,173.1	12.73	3.24	84.41
	model	0.0	1,173.1	4.31	2.00	29.14

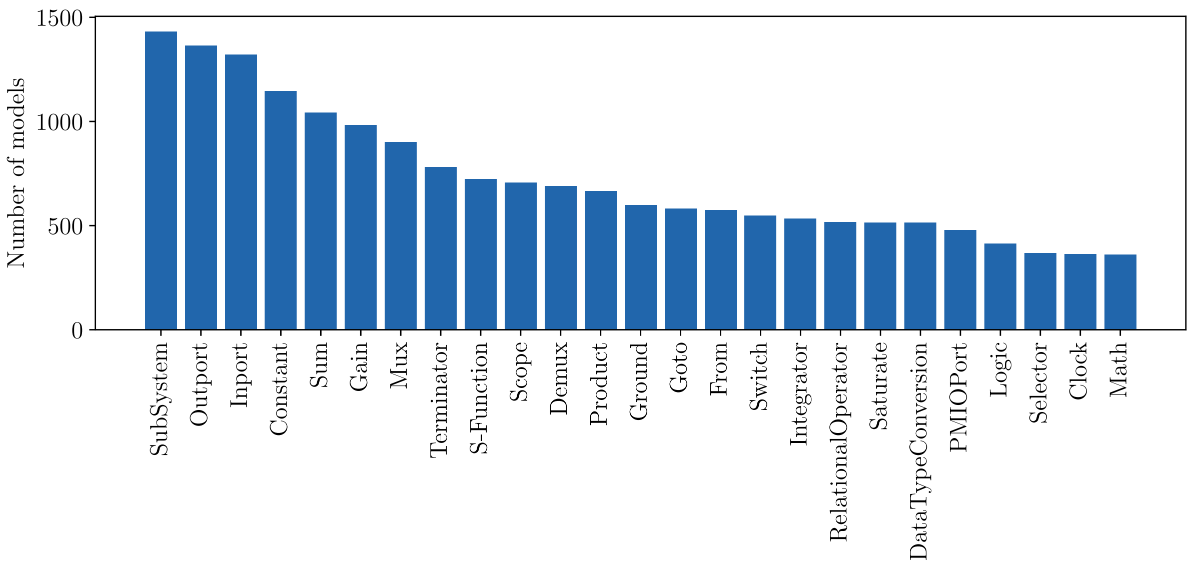


Figure 3.5. The 25 most common block types according to the number of models they are used in.

As with the previously presented metrics, the number of subsystems per project and per model does not show any particular pattern. Again, some very large projects are causing the high maximum value and the high standard deviation.

Similarly, with all of the reported complexity metrics, the mean, median, and standard deviation distribution shows the diversity of models and projects regarding their complexity. The large maximum values of the Cyclomatic Complexity originate from extensive library models from a drive-train simulation. In contrast to these extreme examples, 1088 models exhibit a minimal Cyclomatic Complexity of 1. The Halstead Difficulty values are not as divergent and do not show prominent outliers. However, in addition to the median, the 957 models showing a complexity of 0 signify a large number of low complexity models. The Henry-Kafura Information Flow exhibits one radical outlier with the noted maximum value. The next lowest value amounts to just 13.000. This outlier originates from a demo model contained in a Mathworks tool collection. In contrast to this, there are 1366 models in the dataset with a Henry-Kafura Information Flow of 0. This divergence explains the reported values in the table. Regarding System Complexity, the shown maximum value is an exception in the dataset, with the next highest value amounting to just 178.

Diving deeper into the models' contents, we can see that some block types are commonly used in most models. Figure 3.5 shows the 25 most commonly occurring block types with the number of models they are used. The most used type of block, being the subsystem, highlights the importance of modularization in Simulink projects. Unsurprisingly, in-, and outports, which provide the basic functionality to receive and send signals, are used in most models. The few projects not incorporating these blocks are not used to process data but mainly for simulations using generated signals as inputs and only producing some visualization in scope instead of outputting a signal.

In addition to the models' contents, we analyze the file types of the models within the projects, with the majority of 107 projects (55 %) exclusively containing models with the older file type *.mdl*, 72 projects (37 %) only containing *.slx*-files and 15 (8 %) containing both. 40 % of the models with the newer *.slx* format has been created before 2012 when this file format was first introduced. While appearing counter-intuitive, this finding indicates that these models were created in the previous *.mdl* format but were later transferred into the newer *.slx* format. This hypothesis is further supported by the fact that of these 40 % of *.slx* model files being created before 2012, 95 % were still under active development after introducing the new file format.

Comparison by origin: This comparison analyzes how model size and organization differ between models from different origins, which we have determined in Section 3.1.5. To that end, we group our measurements by the categories Academic, Mathworks, Industry, and No Information, comprising 460, 619, 309, and 346 models, respectively. We use boxplots [32] to illustrate these aggregated values; the orange line within a box represents the median, the size of the box is determined by the first and third quartile, and the whiskers represent the fifth and ninety-fifth percentile. Outliers are displayed as circles.

For most industry category projects, the number of models per project is much lower than in the other categories, as shown in Figure 3.6a. Only the projects with no origin information

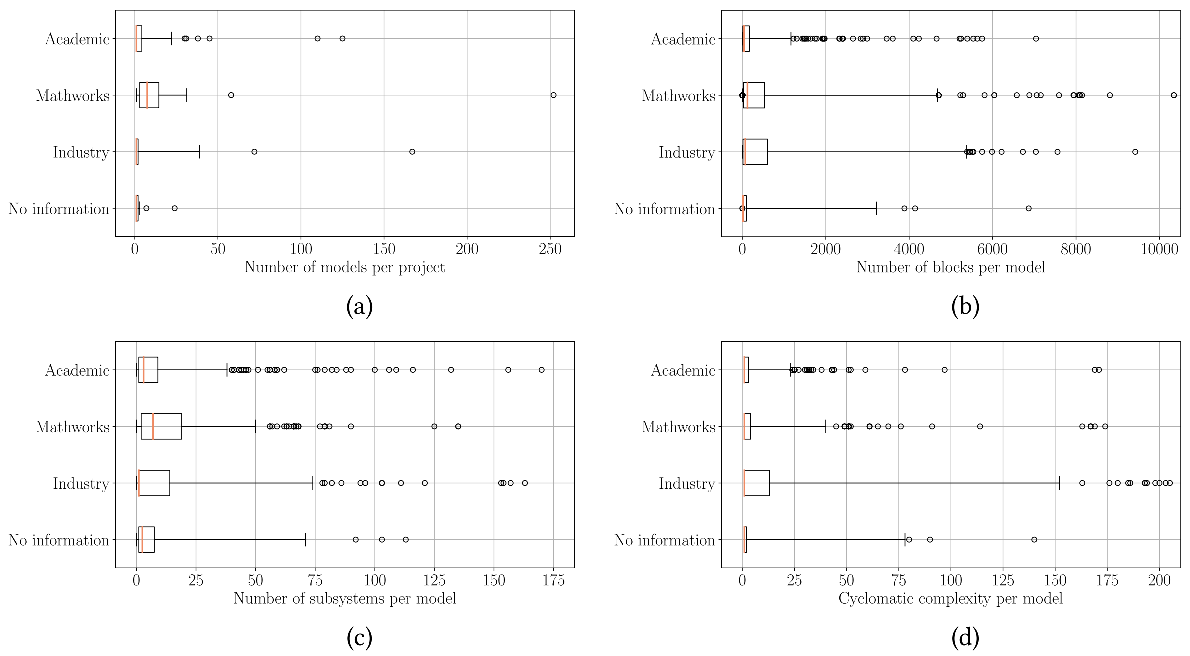


Figure 3.6. Comparison of the different project origins w.r.t (a) the number of models per project, (b) the number of blocks per model, (c) the number of subsystems per model, and (d) the Cyclomatic Complexity per model.

are even smaller. The largest projects in terms of the number of models can be found in the Mathworks category.

When comparing the model metrics based on the project's origin, three metrics stand out and show a difference between origins. Figure 3.6b shows the comparison of origins regarding the number of blocks per model. For reasons of clarity, we cleaned the plot of outliers, *i.e.*, models with more than 10,500 blocks. Therefore, there are eight models in the Mathworks category and three models in the industry and academic categories that are not displayed in the plot. These outliers consist of libraries and large simulation models. Models from the Mathworks category show the largest median, with 123 blocks per model, while the industry models show the largest distribution. While the academic category shows the most narrow distribution of the number of blocks per model, the category also exhibits the most outliers.

Figure 3.6c shows the subsystems per model for all origin categories. This plot is cleaned of outliers with more than 175 subsystems, accounting for one model each in the academic and industry categories. When considering the modularization of models in terms of the contained subsystems, models from the Mathworks category show the most usage of subsystems due to the high median, larger third quartile, and high density of outliers above the 95 percentile. However, there are more models with a larger number of subsystems in the industry category, while the majority of industry models use fewer subsystems than the Mathworks models. Models from academia are least modularized, with just 11 subsystems per model on average.

Lastly, Figure 3.6d shows the Cyclomatic Complexity of models for the different categories. There are four industry models and one from academia not shown in the plot, with values

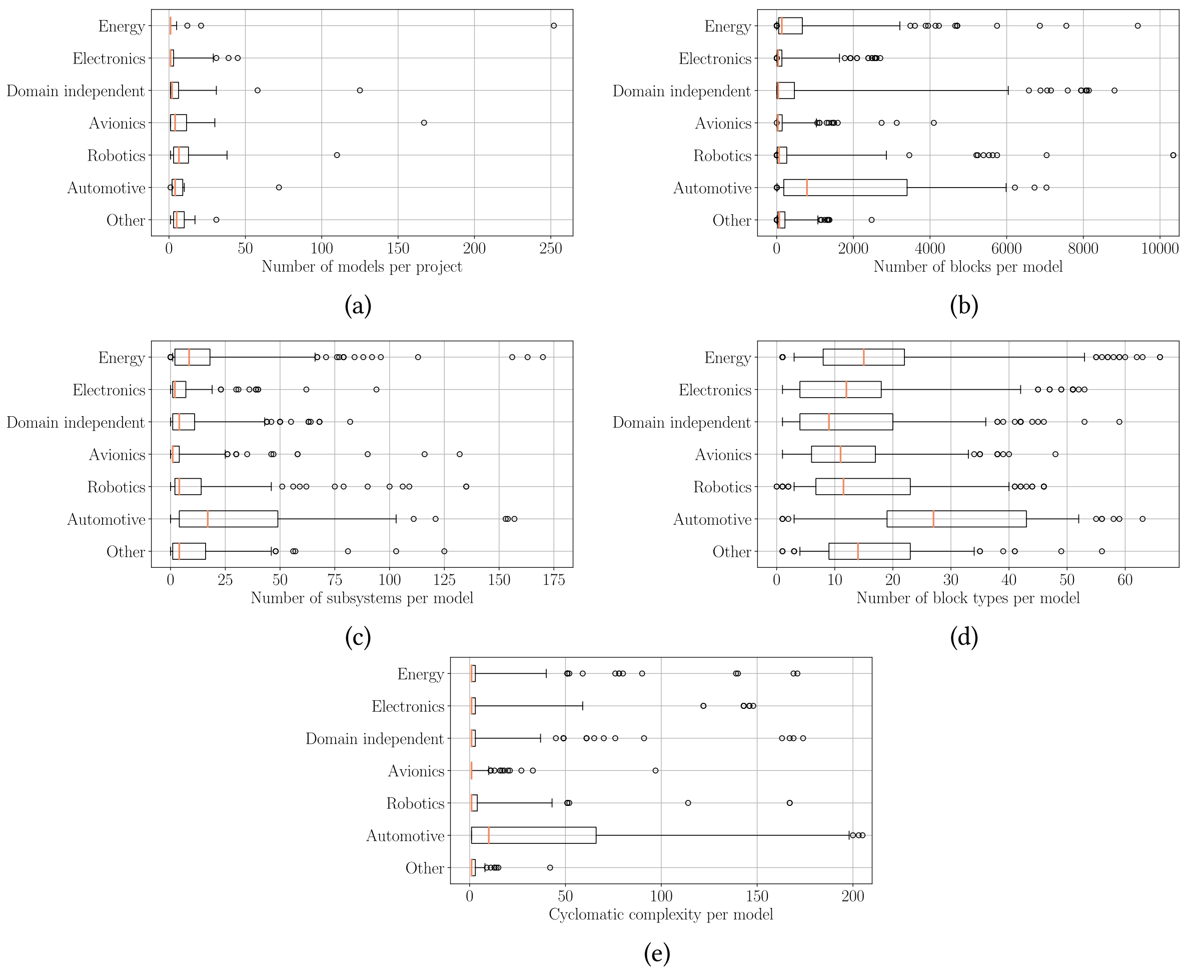


Figure 3.7. Comparison of the different project domains w.r.t. (a) the number of models per project, (b) the number of blocks per model, (c) the number of subsystems per model, (d) the number of different block types per model, and (e) the Cyclomatic Complexity per model.

larger than 210. Interestingly, all categories exhibit a median of 1, as in the overall comparison before. Still, the industry category contains the most complex models, with an average Cyclomatic Complexity of 30. The much larger ninety-fifth percentile also signifies this.

Comparison by application domain: We compare the models concerning their identified application domain from Section 3.1.5. The largest domain in terms of the number of models is the energy domain, with 354 models. The other domains split up as follows: 339 are domain-independent, 280 from electronics, 257 from avionics, 232 from robotics, 151 from several 'other' domains, and 121 in the automotive domain.

Figure 3.7a shows the models' distribution over the projects of the domains. Projects from avionics and robotics domains are the largest in terms of the number of models per project, with 17.1 and 16.6 models per project on average. The smallest projects originate from the electronics and energy domain, with 6.1 and 6.7 models per project on average. The extreme outlier in the energy domain is an extensive model of a wind turbine. Apart from the outliers and the ninety-fifth percentiles, the projects in all domains are rather small, with each of the third quartiles being under 15 models per project.

As in the preceding section, Figure 3.7b shows the blocks per model for the different domains of the projects. This plot is again cleaned off the outliers above 10,500 blocks per model. Therefore not shown are five domain-independent projects, three from the robotics and automotive domains, two from energy, and one from avionics. Figure 3.7b shows that most automotive domain models are significantly larger than in the other domains.

Figure 3.7c shows the number of subsystems per model for the identified domains, excluding one outlier each from the automotive and energy domain. With the automotive domain models being the largest ones, they also show the highest amount of subsystems per model. Therefore, the models in the automotive domain show the highest degree of modularization. The energy domain exhibits the second largest number of subsystems in the models. Generally, the distribution of subsystems over the domains follows a similar trend as the number of blocks.

The number of different block types per model is shown in Figure 3.7d. The automotive domain shows the most diversity in terms of the block types used in the models, with 29 unique block types per model on average. However, many highly diverse models belong to the energy domain, as signified by the ninety-fifth percentile and the outliers. The remaining domains are rather similar in their unique block types per model, exhibiting only minor differences.

Lastly, Figure 3.7e shows the Cyclomatic Complexity of models for the domains. The plot is cleaned of values above 210, which relates to three models from the automotive and two from the energy domain. The previously identified trend holds in this respect as well, as the automotive domain is the most prominent featuring the most complex models. While all other domains exhibit a median of just 1, the automotive domain exhibits a value of 10. Further, all third quartiles are below 5; just the automotive domain exhibits a value of 66 for the third quartile.

***RQ2:** What is the size of the Simulink models and how are they organized within their defining projects?*
The majority of projects and models are rather small. The largest models stem from the automotive domain, exposing a high degree of modularity through subsystems and higher Cyclomatic Complexity. Models originating from industry and Mathworks are the most modularized.

RQ3: Project and Model Evolution

As described in Section 3.1.4, we analyze the subset of projects hosted on GitHub to evaluate evolutionary aspects. In order to evaluate if this subset represents the characteristics of the whole corpus, we analyzed all metrics reported in Section 3.1.5 on this subset as well. While the GitHub subset is missing some of the most extensive projects, identified as outliers in Section 3.1.5, the overall metrics are comparable to those of the whole corpus. The lack of some of the largest projects especially reflects in the metrics for subsystems, Cyclomatic Complexity and System Complexity, where the GitHub subset shows smaller values on average. However, the most extensive projects are outliers of the entire corpus and therefore

Table 3.2. Calculated metrics for the projects' evolution.

Project Metric	Min.	Max.	Mean	Median	Std. Dev.
Number of commits	1.0	589.0	56.8	8.0	120.1
Merge commits in %	0.0	16.9	2.4	0.0	4.5
Number of authors	1.0	16.0	2.7	1.0	3.4
Lifetime in days	0.0	2,273.0	250.8	50.0	511.3
Commits per day	0.005	14.0	1.9	0.5	3.5
Model commits in %	3.1	100.0	44.2	42.9	29.6
Model authors in %	33.3	100.0	82.2	100.0	24.2

Table 3.3. Calculated metrics for the models' evolution.

Model Metric	Min.	Max.	Mean	Median	Std. Dev.
Number of updates	0.0	42.0	2.3	1.0	3.6
Number of authors	1.0	4.0	1.3	1.0	0.5
Abs. lifetime in days*	0.0	2,153.0	204.2	1.0	383.7
Abs. lifetime in days**	0.0	7,071.0	1,350.1	885.0	1,381.1
Rel. lifetime in %*	0.0	100.0	23.9	1.0	33.9

**Calculated for GitHub models based on commit data.*
***Calculated for* all *models of the corpus based on the model files' internal meta data.*

deviate from most corpus projects. Arguably, their lack in the GitHub subset does not harm its representativity.

Evolutionary data from the projects mentioned above can be seen in Table 3.2, whilst the models' evolution characteristics are summarized in Table 3.3. Similar to the results of the static properties of **RQ2** (see Section 3.1.5), these projects and models are highly diverse concerning their evolutionary characteristics, as the standard deviation is bigger than the mean for most metrics. For some metrics, the standard deviation is more than twice the mean value. Further, all metrics' median values are lower than the respective mean values, which indicates that only a few projects are substantially more long-living, more frequently maintained, and have more authors.

Projects: Most projects show a rather small number of commits, as the median only amounts to 8 commits (see Table 3.2). The ratio of merges to all commits in the GitHub projects is even smaller; the median lies at 0 %, with 1.4 merge commits per project on average. Similarly, the number of people actively working on the projects is small, comprising only 2.7 authors on average. The project lifetimes vary greatly between zero days (1 commit only) and more than six years. About half of the projects show an active maintenance span which is less than 50 days. 44.2 % of the commits modify Simulink models, indicating that the models can indeed be considered as primary development artifacts of the projects. The model-author-ratio further supports this: in most cases, all authors of a project also edit the model files, with the mean value being at 82.2 %.

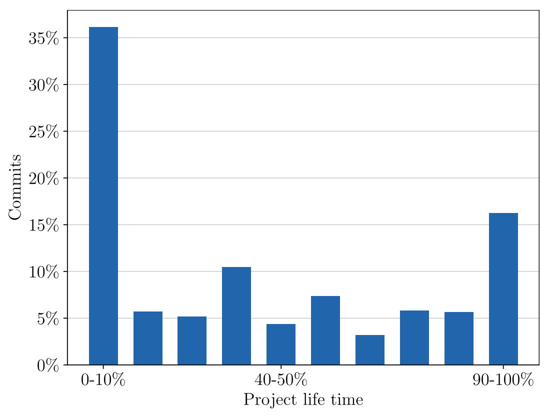
Models: Table 3.3 shows that, on average, a model is updated about two times after its initial creation. For most of the models, these modifications are performed by a single developer since, on average, 1.3 developers contribute to a model over its entire lifetime. The mean time span in which a model is under active development is about 204 days. On the contrary, most models have an active lifetime of only one day – indicated by the median. The relative time a model is actively developed presents a median and mean value of 1 % and 23.9 % of the entire project's lifetime.

Additionally, we evaluated all models' absolute lifetime by analyzing the Simulink model files, which expose their initial creation and last modification dates. The penultimate line of Table 3.3 summarizes lifetime characteristics obtained from Simulink files of 1,686 models in our corpus. For 48 of the models, the format of the dates was corrupted and not retrievable. In particular, it can be seen that the mean and median values differ significantly in comparison to the lifetime calculated for GitHub projects. Further, the most long-living model was under development for almost 20 years. This information might indicate that some models have been developed offline and were later committed and pushed to the central repository for the sake of distribution and archiving. Furthermore, others represented file hosting services like Mathworks and SourceForge started hosting files earlier than GitHub.

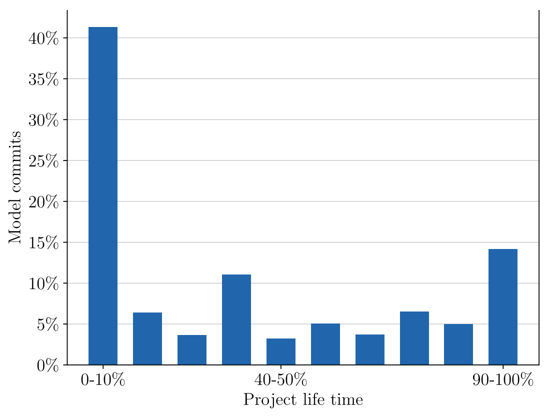
Distribution of development workload over project lifetime: Figure 3.8a shows the distribution of project commits over a project's lifetime, averaged over all projects. As the median project lifetime is about 50 days, each bin represents five or more days for most projects. Apart from bursts of development activity at the beginning and the end of a project, a rather even distribution of project commits is observed. The burst in the first tenth of a project's lifetime is dominating with 36 % of all commits falling into this initial period. A similar pattern can be seen in Figure 3.8b, which depicts commits' distribution, that modify a Simulink model, again, averaged over all projects.
From comparing Figure 3.8a and Figure 3.8b, we can conclude, that the overall workload on models and the rest of the project is similarly distributed. A minor difference can be observed concerning the bursts at the beginning and the end of a project, where the first burst of development activity is even more distinct for model commits than for project commits.

Figure 3.8c shows a nearly identical graph to Figure 3.8b, as it plots the distribution of committed Simulink model modifications over a project's lifetime. The difference is, that Figure 3.8b counts a commit with at least one created or updated model, and Figure 3.8b counts each created or updated model, individually. As the graphs are extremely similar, it follows that the average commit on one or more models over a project's lifetime modifies the same amount of models.

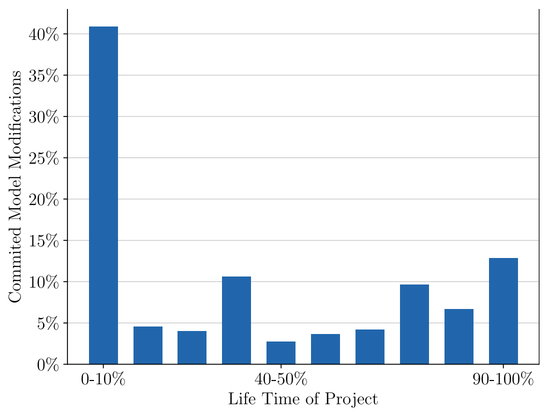
Figure 3.8d shows the ratio of models under active development during a project's lifetime, averaged over all projects. A model is counted in each bin that falls between its first commit and last commit: suppose a model is created at the very start of the project and last modified just before the project's half time, then it will be counted in the first four bins separately. It can be seen that 52 % of the models of the GitHub subset are created in the first 10 % of the project lifetime. At least half of these models were never modified again, as the second bin



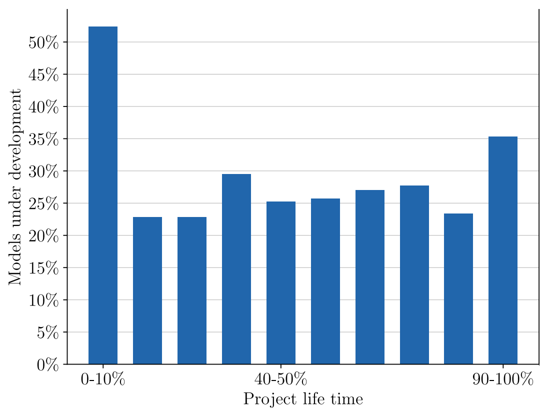
(a)



(b)



(c)



(d)

Figure 3.8. Figure (a) shows the distribution of commits over a project's lifetime. Figure (b) shows the distribution of commits, that modify Simulink models over a project's lifetime. Figure (c) shows the distribution of committed Simulink modifications over a project's lifetime. Figure (d) shows the *ratios* of models under development over a project's lifetime. Note that a model will be counted in every bin from its first commit till its last in (d).

only holds 23 % of models that are under active development. Some models are created only in the last 10 % of a project's lifetime. On average, more than 22 % of the models are under active development during the entire duration of a project. Again, this can be interpreted as an indicator that the models within a Simulink project can be considered primary development artifacts since there are no project phases in which the models are not edited.

RQ3: *How do Simulink projects and their models evolve over time?*
35 projects from our corpus are hosted on GitHub, most of which are under active development for less than 50 days. The median project receives a commit every second day. Most models are rarely updated and commonly maintained by only a single developer. Bursts of commit activity can be seen at the beginning and the end of a project, with workload regarding models following a similar trend. More than a fifth of the average project's models are under active development throughout the entire lifetime of a project.

3.1.6 Threats to Validity

We discuss potential threats to our study results' validity, using the scheme established by Wohlin et al. [122].

Internal Validity

Threats to internal validity are related to our methodology's potential systematic errors, most notably concerning the collected and analyzed data.

Some of the classifications for ***RQ1*** (origin and application domain) have been done manually, which may be biased by the subjective assessment of individual researchers or by simply overlooking relevant information. To mitigate this bias, we formed a team of four researchers to rate any manual classification task results. Their inter-rater agreement was good, as reported in Section 3.1.5. If two or more researchers were unsure how to assess a project, we abstained from making a final yet potentially misleading decision and classified the project as *unclear.*

The reported quantitative measures are calculated using scripts that we developed as part of this study. We took several countermeasures to rule out potential errors in these calculations. For those metrics already reported in the study of Chowdhury et al. [236], which are based on a model corpus that overlaps with ours, we have checked the plausibility and were able to reproduce their results. Further, we used the Matlab/API to parse the Simulink models, which prevents errors introduced by other custom-built Simulink parsers. We checked the results of our scripts on sample models of the corpus to assure correctness. Since our automated check on TargetLink and Embedded Coder usage did not result in any findings, we used example models for code generation provided by Matlab Simulink also. Our scripts successfully detected indications of code generation in these models. For the evolutionary metrics extracted from GitHub, we used an established tool, namely PyDriller [253], to extract meta-data from the commit history.

Additionally, in our analysis, we focused only on the information that is provided directly in the Simulink models, as it is also done in other studies [127, 196, 236, 291]). However, Simulink models can also reference Matlab code and functions. Arguably, these are a part of the model, and most repositories do not contain just Simulink models. The analysis of Matlab code was out of the scope of our study and scripts. Our results may be affected by this threat if, for example, most of the complexity resides in the outsourced Matlab code instead of the Simulink model.

The information presented in this study is limited to the data available in the project repositories. To not miss anything, we performed manual inspection of the models in a open coding fashion [98] (see Section 3.1.4). However, if relevant information about the project is not reflected in the repositories or their meta-information, we may have missed it.

Construct Validity
The construct validity concerns whether the study answers the posed research questions.

We investigated the projects and models curated in our corpus from three perspectives (context, size/organization, and evolution). However, the project and model characteristics explored for each of these perspectives are not meant to be comprehensive. Though we selected these characteristics based on existing guidelines for empirical research and related work, future empirical studies may aim for different characteristics that are not yet considered by our analysis.

Moreover, our classification scheme for ***RQ1*** may be incomplete. However, the main reason that projects could not be assigned to a dedicated origin or application domain was missing information. No additional category arose during the discussion of the researchers who did the manual classification.

Regarding the evolution of projects, we rely on the commit history of GitHub. However, as with any repository, we do not have further information regarding subordinate processing of the models between commits or before the first commit. Thus, we cannot assess whether there is an underlying modeling process apart from the explicit repository commits. In particular, mainstream version control systems such as GitHub are file-based and work on a textual or binary representation of the managed artifacts, which is still considered an obstacle for the versioning of models. Thus, version control systems are not as integrated into the development process as it is the case for code-centric development.

External Validity
The external validity pertains to the question to which extent our results are generalizable. Raw data used in our study is taken from a limited set of data sources, namely a publicly available corpus of Simulink models and according meta-data extracted from the subset of GitHub projects for answering ***RQ3***. We did not do any systematic mining of open source platforms (*e.g.*, GitHub, SourceForge, BitBucket) beyond the projects included in the corpus. More specifically, we cannot claim that the analyzed corpus is statistically representative for the population of all existing Simulink projects (not even for the publicly available ones). The reader should have in mind that access to a comprehensive population list regarding *all publicly available open-source Simulink projects and models* is impossible; thus, we cannot infer that any accessible population is "representative". However, the selected corpus provides the currently largest [291] and publicly available set of open source Simulink models from hosts like Mathworks, SourceForge, GitHub, and other web pages, and it even includes two other compiled corpora [148, 290]. This makes us confident that our results generalize to other open-source Simulink projects.

Conclusion Validity
Conclusion validity pertains to the degree to which we can be sure that our conclusions are reasonable.

Due to the lack of reliable indicators, our study does not capture the intent behind creating a model, yielding a spectrum that heavily influences a model's characteristics. On the one hand of this spectrum, there are simple example models that are deliberately kept as tiny as possible, *e.g.*, for the sake of teaching. On the other hand of the spectrum, some models are created to model a real-world system or phenomenon, thus growing in size and complexity. Not distinguishing the models by their intent of creation may lead to the fact that a few outliers dominate many of the aggregated metrics presented in Table 3.1 at both ends of this spectrum. A classification of the intent could help eliminate such outliers and get a better picture of the models within each intent category. However, the aggregated values are not meant to characterize a specific class of models, but are calculated for characterizing our entire corpus.

Instead of the intent behind creating a model, we classified our projects and models according to their origin and application domain (see Section 3.1.4), which we use to get an overview of the characteristics in each of these categories (see Figure 3.6 and Figure 3.7). The results may be biased by intents not equally distributed over the models' origins and application domains. For example, by chance, it might be the case that most of the models in one application domain are toy examples created for the sake of teaching, while the models in another domain are representing abstractions of real-world systems. Again, a classification by intent could help to rule out such undesired effects.

3.1.7 Discussion: Suitability for Empirical Research

In this section, we discuss our findings together with the opinion of a Simulink expert. The expert, a partner in the context of a research project, has more than six years of experience in developing a quality assurance and optimization tools for Simulink models. In his work, he is confronted with many Simulink models and projects in various stages of development and from different application domains, including the automotive, automation, and lift domains. Please note that this consultation of an expert is not meant to be part of our research methodology as the expressiveness of just one expert's opinion is somewhat limited. However, we included the expert to help us form our interpretation less subjectively and become more informed from a practitioner's point of view. The talk with the expert started with presenting our results, followed by an open discussion on the project and model characteristics. The goal was to get an expert opinion on the general suitability of our corpus.

Suitability from the Perspective of Context

Although our corpus comprises Simulink projects of various origins, most of them have an academic background (58 %), whilst only 13 % originate from the industry. However, Figures 3.6a to 3.6c show that models from academic and industrial projects are not that different. Moreover, several different application domains are represented within our corpus, although the distribution is skewed towards the energy and electronics domains while others are missing (*e.g.*, defense and automation).

According to the interviewed Simulink expert, the automotive domain seems to be underrepresented within our corpus. The expert was also surprised that we did not find any indication of code generation in our models. According to the practitioner experience, industrial models are usually used for generating code, which is eventually deployed on some hardware. The expert expected more models employing Embedded Coder. He assumes that TargetLink is not used in the corpus' models because it is too expensive for open-source Simulink development.

In conclusion, our corpus does show some variation in terms of the project context. However, the corpus should not be used to compare characteristics between different domains since not all domains are covered, and the distribution is imbalanced. Additionally, the corpus is not suitable for studies on code generation since we found no evidence of code generation capabilities in the projects.

Suitability from the Perspective of Model Size and Organization

The Simulink expert reveals that most models in our corpus are considerably smaller (median of 70 blocks per model) than the average models from the industry (his estimation is a median of about 1,000 blocks per model). Typical industry models he analyzes consist of 200 – 2,000 blocks, only a few exceed 20,000 blocks. Thus, the largest models in our corpus are comparable to large models in the industry. The expert also confirmed that the distribution of block types in our corpus is similar to industrial models. He was surprised by the ratio of blocks to the subsystem. In his opinion, this relation indicates rather mature models in terms of modularization.

As presented in Section 3.1.5, the corpus's diversity may be useful for testing and validating tools or automated approaches. The variety of models can cover a vast spectrum of test cases. Further, for studies with more specific requirements towards certain model characteristics, the corpus can be leveraged to produce a subset under the application needs (e.g., only large models, only models with many subsystems, exclusion of library models).

In terms of model size and organization, the corpus is well suited for testing and evaluating tools. The enclosed models exhibit a wide range of characteristics. The largest ones are comparable to industry models in terms of size, which is especially suitable for scalability and performance tests. On the other hand, large models still are a minority. Therefore, the corpus may not be suitable for applications with especially high prerequisites concerning the amount of data, *e.g.*, machine-learning approaches.

Suitability from the Perspective of Project and Model Evolution

Our corpus provides limited opportunities to research projects and model evolution. Only 35 of the 194 projects are hosted on GitHub and offer the full project commit history. The results presented in Section 3.1.5 show that most projects are rather short-lived (<50 days) and are maintained by only one developer. A low number of merge commits also indicates little collaborative work available for analysis. However, a few projects in our corpus provide opportunities to study their evolution (*e.g.*, in a case study research). For example, a NASA project[12] is active for 2,273 days, a Mathworks Simulink tools project[13] has 589 commits, and a driving chair simulator[14] has 16 developers. Moreover, with, on average, 44 % of commits affecting models, the development of most projects indeed focuses on Simulink models.

Despite the cases mentioned above, we conclude that most projects are not suitable for empirical studies from an evolutionary perspective, confirmed by the Simulink expert.

[12] https://github.com/nasa/T-MATS (visited on 15/12/2025)
[13] https://github.com/analogdevicesinc/MathWorks_tools (visited on 15/12/2025)
[14] https://github.com/Alexanderallenbrown/MotionBase/wiki (visited on 15/12/2025)

According to him, the evolution characteristics extracted from GitHub do not mirror the evolution of industrial Simulink projects. Typically, more developers are involved in a project, and the number of commits steadily increases towards the end of the project or a release.

3.1.8 Conclusions

In this paper, we collect and investigate a set of 1,734 freely available Simulink models from 194 projects and analyze their basic characteristics and suitability for empirical research. Our analyses regarding project context, size and organization, and evolution have shown that the projects and models are highly diverse in all aspects.

In principle, the models in our corpus are suitable for empirical research. Depending on the research goals, the subsets of the corpus might have to be selected. According to the Simulink expert, many corpus models can be considered mature enough for quality analysis purposes. Another use case might be unit testing, as many test cases can be covered with a diverse set of models. Generally, the usage of a publicly available model corpus or a subset enables researchers to reproduce findings, publish subsequent studies, and use them for validation purposes.

For other kinds of empirical research, however, our corpus might be of limited value. Most industry models use code generation at some development stage, which is not represented in the corpus at all. Domain-wise, the corpus is skewed towards the energy sector. Run-time analysis with big models (*e.g.*, 100k blocks or more) is possible with only a few models. Many projects are no longer under active development or maintenance, which may be necessary for testing up-to-date Simulink versions and newer features or consulting the developers involved in a Simulink project.

In the future, we want to investigate the models' contents and their evolution. To understand their basic characteristics, most of our current metrics refer to the models' size and basic organization within projects, which could be complemented by structural complexity metrics or even qualitative analyses in the future. The evolutionary characterization might be worth examining content-related characteristics such as structural differences between versions, complementing our high-level analyses of the development history. Acquiring a bigger set of Simulink projects from GitHub akin to the method used in [282] promises to gain more generalizable statements about ***RQ3***.

3.2 Case Study 1: RaQuN: A Generic and Scalable N-Way Model Matching Algorithm

↪ Thesis Roadmap

Authors: Alexander Schultheiß, Paul Maximilian Bittner, Alexander Boll, Lars Grunske, Thomas Thüm, Timo Kehrer.
Published in: Software and Systems Modeling.
Copyright: © 2022 The Authors.
Extends: Scalable N-Way model matching using multi-dimensional search trees, Alexander Schultheiß, Paul Maximilian Bittner, Lars Grunske, Thomas Thüm and Timo Kehrer. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), 2021.

Abstract: Model matching algorithms are used to identify common elements in input models, which is a fundamental precondition for many software engineering tasks, such as merging software variants or views. If there are multiple input models, an n-way matching algorithm that simultaneously processes all models, typically produces better results than the sequential application of two-way matching algorithms. However, existing algorithms for n-way matching do not scale well, as the computational effort grows fast in the number of models and their size. We propose a scalable n-way model matching algorithm, which uses multi-dimensional search trees for efficiently finding suitable match candidates through range queries. We implemented our generic algorithm named RaQuN (**Ra**nge **Qu**eries on **N** input models) in Java, and empirically evaluate the matching quality and runtime performance on several datasets of different origin and model type. Compared to the state-of-the-art, our experimental results show a performance improvement by an order of magnitude, while delivering matching results of better quality.

3.2.1 Introduction

Matching algorithms are an essential requirement for detecting common parts of development artifacts in many software engineering activities. In domains where model-driven development has been adopted in practice, such as automotive, avionics, and automation engineering, numerous model variants emerge from cloning existing models [106, 128, 211]. Integrating such autonomous variants into a centrally managed software product line in extractive software product-line engineering [125] requires to detect similarities and differences between them, which in turn requires to match the corresponding model elements of the variants. Moreover, matching algorithms are an indispensable basis for merging parallel lines of development [48], or for consolidating individual views to gain a unified perspective of a multi-view system [60].

Currently, almost all existing matching algorithms can only process *two* development artifacts[31, 47, 53, 55, 57, 63, 66, 68, 73, 78, 123, 147], whereas the aforementioned activities typically require to identify corresponding elements in *multiple* (*i.e.*, $n > 2$) input models. A few approaches calculate an n-way matching by repeated two-way matching of the input artifacts [118, 130, 168, 202, 203, 305]. In each step, the resulting two-way correspondences are simply linked together to form *correspondence groups* or *matches* (aka. *tuples*[135]).

However, sequential two-way matching of models may yield sub-optimal or even incorrect results because not all input artifacts are considered at the same time [135]. The order in which input models are processed influences the quality of the matching because better match candidates may be found after an element has already been matched. An order might be determinable if a reference model is given, but this is typically not the case [60, 170, 184, 203, 206, 280, 287]. An optimal processing order cannot be anticipated and applying all $n!$ possible orders for n input models is clearly not feasible [168].

The only matching approach which simultaneously processes n input models is a heuristic algorithm called NwM by Rubin and Chechik [135]. NwM delivers n-way matchings of better quality than sequential two-way matching. However, we faced scalability problems when applying NwM to models of realistic size, comprising hundreds or even thousands of elements.

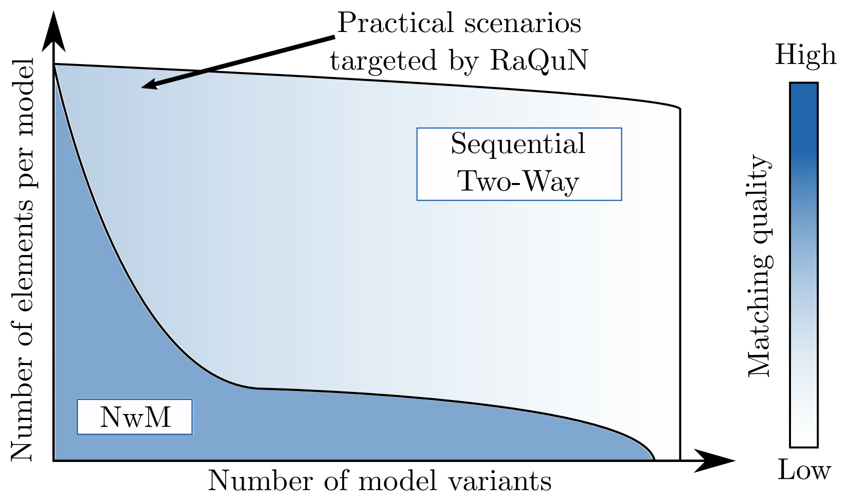


Figure 3.9. Symbolic illustration of the limitations of existing n-way matching solutions and the need for further research.

The most likely reason for this is the required number of model element comparisons, which often leads to performance problems even in the case of few input models if these models are large [72, 82, 111].

The limitations of existing solutions are symbolically illustrated in Figure 3.9. By applying sequential two-way matching, n-way matching can be done for both large models and large sets of models, but scalability comes at the price of quality. NwM delivers n-way matchings of better quality, but does not scale for large models, even if the number of model variants is limited to only a few. Thus, there is a strong need for a scalable n-way matching solution.

In our MODELS paper [324], we proposed RaQuN (**Ra**nge **Qu**eries on **N** input models), a generic, heuristic n-way model matching algorithm. As illustrated in Figure 3.9, RaQuN targets practical scenarios in which several models are to be matched that each comprise a large number of elements. The key idea behind RaQuN is to map the elements of all input models to points in a numerical vector space. RaQuN embeds a multi-dimensional search tree into this vector space to efficiently find nearest neighbors of elements, *i.e.*, those elements which are most similar to a given element. By comparing an element only with its nearest neighbors, RaQuN can reduce the number of required comparisons considerably. For our empirical assessment, we used datasets from different domains and development scenarios. Next to academic and synthetic models [115, 135], we investigated variants generated from model-based product lines [95, 112, 158, 220], and reverse-engineered models from clone-and-own development [128, 136]. Our evaluation showed that RaQuN reduces the number of required comparisons by more than 90% for most experimental subjects, making it possible to match models of realistic size simultaneously.

In this paper, we extend our previous publication [324] in three aspects. First, we evaluate RaQuN on five additional experimental subjects comprising Simulink models, a model type which we have not considered before. Second, we propose two additional configuration options for RaQuN's configuration points (see Section 3.2.4); the first option targets the reduction of RaQuN's runtime (see Section 3.2.4), and the second option the improvement of

RaQuN's matching quality with respect to precision and recall (see Section 3.2.4). Finally, we extend our evaluation of the impact of RaQuN's configuration points on runtime and matching quality in terms of two additional research questions (see RQ1 and RQ3 in Section 3.2.5).

In summary, our contributions are:

Generic Matching Algorithm (Section 3.2.3). We present a generic simultaneous n-way model matching algorithm, RaQuN, that uses multi-dimensional search trees to find suitable match candidates.

Domain-agnostic Configuration (Section 3.2.4). For all variation points of the generic algorithm, we propose domain-agnostic configuration options turning RaQuN into an off-the-shelf n-way model matcher.

Empirical Evaluation (Section 3.2.5). We show that RaQuN has good scaling properties and can be applied to large models of various types, while delivering matches of better quality than current state-of-the-art approaches.

3.2.2 N-Way Model Matching

In this section, we illustrate the n-way model matching problem with a simple running example, and discuss how algorithmic approaches calculate a matching in practice. As our running example, we consider the three UML class diagrams A, B, and C given in Figure 3.10, which are fragments of the hospital case study [115, 135]. Each of the three models is an early design variant of the data model of a medical information system. We use the symbolic identifiers 1 to 8 to uniquely refer to the models' classes.

Our representation of models follows the so-called *element-property approach* [135]. A model M of size m is a set of *elements* $\{e_1, \dots, e_m\}$. Each model element $e \in M$, in turn, comprises a set of *properties*. For our running example, we consider UML classes as elements, and we restrict ourselves to two kinds of properties, namely class names and attributes. However, the element/property approach is general enough to account for other kinds of model elements (*e.g.*, states and transitions in state-charts), and other kinds of properties (*e.g.*, element references or element types).

Intuitively, n-way matching refers to the problem of identifying the common elements among a given set of n input models. A reasonable matching for our example is illustrated in Figure 3.10, indicated by solid lines. The models A and B each contain a class named *Physician*. Both classes have several attributes in common and may thus be considered to represent "the same" conceptual model element in different variants. Following common terminology from the field of two-way matching, we say that class *Physician* in model A *corresponds to* class *Physician* in model B. Similarly, each of the three models contains a class named *AdminAssistant*, and all three variants of the class share several identical attributes. Thus, these classes form a so-called correspondence group (aka. *tuple* [135]). We call such a correspondence group a *match*.

Formally, we define an n-way matching algorithm as a function which takes as input a set $\mathcal{M} = \{M_1, ..., M_n\}$ of input models and returns a *matching* T. A matching $T = \{t_1, \dots, t_k\}$ is defined as a set of matches, where each match $t \in T$ is a non-empty set of model elements. Analogously to all existing approaches to n-way matching [118, 130, 135, 168, 202, 203, 305] ,

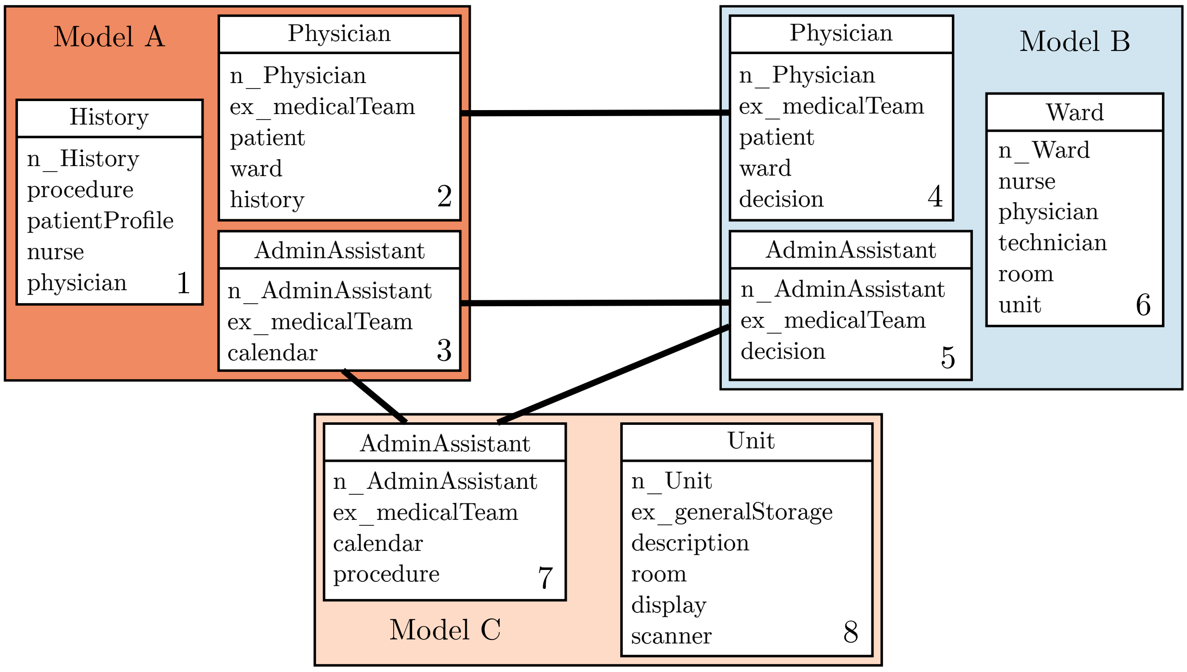


Figure 3.10. Three UML models representing early design variants of the data model of a medical information system, serving as running example.

we assume matches in T to be mutually disjoint, and that no two elements of a match belong to the same input model. Formally, a match t is valid if it satisfies the condition

$$t \neq \emptyset \quad \wedge \quad |t| = |\mu(t)| \tag{3.1}$$

where $\mu(t)$ denotes the set of input models from which the elements of t originate. The intuitive matches illustrated in Figure 3.10, *i.e.*, $\{3, 5, 7\}$, $\{2, 4\}$, $\{1\}$, $\{6\}$, and $\{8\}$, are valid and mutually disjoint.

In theory, a matching could be computed by considering all possible matches for a set of input models. However, this approach is not feasible, as the number of possible matches for a set of models is equal to $(\prod_{i=1}^{n}(m_i + 1)) - 1$ [135], where n denotes the number of models, and m_i denotes the number of elements in the *i-th* model.

A trivial approach would be to rely on persistent identifiers or names of model elements. The limitations of such simple approaches have been extensively discussed in the literature on two-way matching [47, 55, 72, 82, 111] (see related work in Section 3.2.6), and also apply to the n-way model matching problem. Reliable identifiers are hardly available across sets of variants, and names are not sufficiently eligible for taking an informed matching decision without considering other properties. In particular, names are not necessarily unique, and some model elements do not have names at all [160].

In practice, matching algorithms thus operate heuristically. This requires a notion for the *quality* of a match, or in other words, a measure for the similarity of matched elements. Given a match $t \in T$, a *similarity function* calculates a value representing the similarity of the elements in t. We assume that a similarity function makes it possible to (i) establish a partial order on a set of matches, and (ii) determine whether a set of candidate elements should be matched. An example for a similarity function is the weight metric introduced by Rubin and Chechik [135] (see Section 3.2.4).

3.2.3 Generic Matching Algorithm

Algorithm 1 RaQuN

```
1: procedure RaQuN($\mathcal{M}$)                          ▷ A set of input models
2:     $E \leftarrow \bigcup_{i=1}^{i=N} M_i$               ▷ Phase 1: Candidate Initialization
3:     tree ← $createEmptyTree()$
4:     for $e \in E$ do
5:         $v_e \leftarrow vectorize(e)$
6:         tree ← $insert(\text{tree}, e, v_e)$
7:     end for
8:     $P \leftarrow \emptyset$                              ▷ Phase 2: Candidate Search
9:     for $e \in E$ do
10:        Nbrs ← $neighborSearch(\text{tree}, e)$
11:        for nbr ∈ Nbrs do
12:            $p \leftarrow \{e, \text{nbr}\}$
13:            if $isValid(p)$ then
14:                $P \leftarrow P \cup \{p\}$
15:            end if
16:        end for
17:    end for
18:    $\hat{P} \leftarrow filterAndSort(P)$                ▷ Phase 3: Matching
19:    $T \leftarrow \big\{\{e\} \mid e \in E\big\}$
20:    for $\{e, e'\} \in \hat{P}$ do
21:        $t \leftarrow$ select $t \in T$ for which $e \in t$
22:        $t' \leftarrow$ select $t' \in T$ for which $e' \in t'$
23:        $\hat{t} \leftarrow t \cup t'$
24:        if $isValid(\hat{t})$ and $shouldMatch(t, t', e, e')$ then
25:            $T \leftarrow \big(T \setminus \{t, t'\}\big) \cup \{\hat{t}\}$
26:        end if
27:    end for
28:    return $T$                                            ▷ The calculated matching
29: end procedure
```

In this section, we first describe our generic n-way matching algorithm RaQuN (Algorithm 1), followed by an illustration applying the algorithm to our running example introduced in Section 3.2.2, and closing with a theoretical analysis of the algorithm's runtime complexity. We focus on the high-level steps that are performed by the algorithm, while we discuss the details of how each step can be configured later in Section 3.2.4.

Description of the Algorithm

RaQuN takes as input a set $\mathcal{M} = \{M_1, ..., M_n\}$ of n input models and returns a set T of matches (*i.e.*, a matching). The algorithm is divided into three phases. The goal of the first two phases (candidate initialization and candidate search) is to reduce the number of comparisons required in the third phase (matching).

***Candidate Initialization (Line 2–7)*:** In the first phase, RaQuN constructs a multi-dimensional search tree comprising all the elements of all input models as numerical vector representations. First, RaQuN collects the elements of all input models in an element set E, and initializes an empty tree. For each element $e \in E$, a vector representation v_e is determined and inserted into the tree. Hereby, each element is mapped to a specific point in the tree's vector space.

***Candidate Search (Line 8–17)*:** In the second phase, RaQuN determines promising match candidates by considering elements that are close to each other in the vector space, as determined by a suitable distance metric (*e.g.*, Euclidean distance). More specifically, RaQuN retrieves the k' nearest neighbors Nbrs for each element $e \in E$ in the vector space through a k'-NN search on the tree[24]. For every neighbor nbr $\in$ Nbrs of e, RaQuN creates an unordered pair $p = \{e, \text{nbr}\}$. If p is a valid match according to Equation 3.1 (*i.e.*, the two elements belong to different models), p is added to the match candidates P.

***Matching (Line 18–27)*:** In the third and last phase, RaQuN matches elements to each other by comparing the elements in the pairs P directly. First, in Line 18, all candidate pairs in P are sorted descendingly by their similarity, yielding list $\hat{P}$, omitting pairs with no common properties. Next, RaQuN creates a set T of matches such that each element $e \in E$ appears in exactly one single-element match $\{e\}$. The set T is a valid matching in which none of the elements has a corresponding partner. For every candidate pair $p \in \hat{P}, p = \{e, e'\}$, RaQuN selects the two matches t and t' from T which contain the two elements e and e', respectively. Since every element $e \in E$ is in exactly one match in T, the selection of t and t' is unique. If the union $\hat{t} = t \cup t'$ is a valid match and its elements form a good match according to *shouldMatch*, RaQuN updates the matching T by replacing the two selected matches t and t' with $\hat{t}$. The algorithm terminates once all pairs in $\hat{P}$ have been processed. Each match now contains between one and n elements, and T represents a valid matching.

Exemplary Illustration

We illustrate RaQuN by applying it to our running example shown in Figure 3.10, comprising the input models: $\mathcal{M} = \big\{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8\}\big\}$.

***Candidate Initialization*:** RaQuN creates the set of all elements $E = \{1, 2, 3, 4, 5, 6, 7, 8\}$ by forming the union over the models in $\mathcal{M}$. For our example, we choose a very simple two-dimensional vectorization. The first dimension is the average length of an elements' property names, and the second one is the number of properties of an element. Class *1:History-A*, for example, has an average property name length of 9.2 and five properties in total; its vector representation is (9.2, 5). Figure 3.11 visualizes the resulting k-dimensional vector space

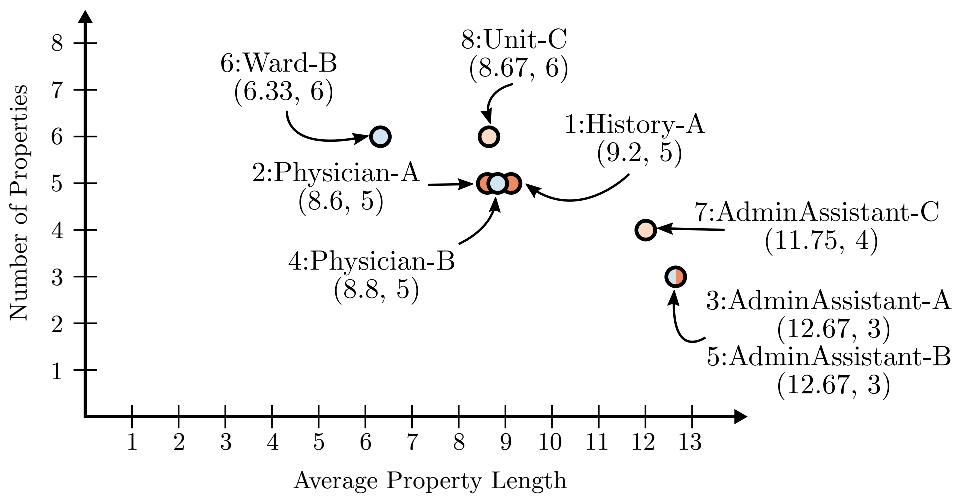


Figure 3.11. Model elements of our running example mapped to points in a k-dimensional vector space (with k=2).

(k=2) and the points of all elements in E. We can see that intuitively corresponding classes are mapped to points close to each other, such as the two 'Physician' classes from models A and B.

Candidate Search: RaQuN performs range queries on the tree to find possible match candidates. For our example, we assume that the candidate search is configured to search for the three nearest neighbors of each element (k'=3). It is possible that multiple elements have the same vector representation and are mapped to the same point in the vector space, such as elements 3 and 5 in Figure 3.11. Therefore, RaQuN might retrieve more than k' neighboring elements. In our example, RaQuN finds the neighbors $\{2, 4, 1\}$ for element *2:Physician-A*, and the neighbors $\{3, 5, 7, 1\}$ for element *3:AdminAssistant-A*. Neighbors forming a valid match with the initial element can be considered as match candidates. For *3:AdminAssistant-A*, the retrieved candidate pairs are $\{3, 5\}$ and $\{3, 7\}$. Once the candidate search has been completed for all elements, we obtain the set P of candidate pairs:

$$P = \Big\{\{1,4\},\{2,4\},\{3,5\},\{3,7\},\{5,7\},\{5,1\},\\ \{6,2\},\{6,8\},\{7,1\},\{8,2\},\{8,4\}\Big\}.$$

Matching: RaQuN sorts the match candidates P by descending confidence whether their elements should be matched, according to its similarity function. For the sake of illustration, we choose a straightforward similarity function: the ratio of shared properties to all properties in the two elements - known as the Jaccard Index [20]. We receive the following (partially) sorted list of candidate pairs:

$$\hat{P} = \Big(\{3,7\}{:}\tfrac{3}{4}, \{2,4\}{:}\tfrac{4}{6}, \{3,5\}{:}\tfrac{2}{4},\\ \{5,7\}{:}\tfrac{2}{5}, \{7,1\}{:}\tfrac{1}{8}, \{6,8\}{:}\tfrac{1}{11}\Big),$$

where $\{x, y\}$:z denotes a pair with elements x and y having a similarity of z. Pairs with a similarity of 0 are removed during sorting, as their elements have no common properties.

Next, RaQuN initializes the set of matches T such that there is exactly one initial match for each element: $T = \big\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}\big\}$. RaQuN now iterates over the pairs in $\hat{P}$ and merges the corresponding matches in T accordingly. To keep the example simple, we assume that matches should be merged if the similarity of the candidate pair is at least $\frac{1}{2}$. The first pair that is selected is $\{3, 7\}$, as its elements have the highest similarity. Thus, RaQuN selects the matches $t = \{3\}$ and $t' = \{7\}$ from T and check whether their comprised elements should be matched. This is the case for the selected matches since the similarity between its elements is $\frac{3}{4} > \frac{1}{2}$. RaQuN thus merges the matches to the new match $\hat{t} = \{3, 7\}$. RaQuN replaces t and t' with $\hat{t}$, and receive $T = \big\{\{1\}, \{2\}, \{3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}\big\}$. In the second iteration, RaQuN selects $t = \{2\}$ and $t' = \{4\}$. Both are merged to the valid match $\hat{t} = \{2, 4\}$. RaQuN repeats this process until all candidate matches in $\hat{P}$ have been considered. We obtain the final matching $T = \big\{\{1\}, \{2, 4\}, \{3, 5, 7\}, \{6\}, \{8\}\big\}$, which is equal to the intuitive matching illustrated in Figure 3.10.

Worst-Case Complexity

We estimate RaQuN's worst-case runtime complexity for each phase. Let n denote the number of input models and m the number of elements in the largest model.

***Candidate Initialization*:** Each element $e \in E$ with $|E| \leq nm$ is vectorized and inserted into the tree. We assume that vectorization is an $O(1)$ operation. Given that insertion into a search tree is possible in $O(nm)$ [24], the worst-case runtime complexity of this phase is $O(nm \cdot (1 + nm)) = O(n^2m^2)$.

***Candidate Search*:** For each of the at most nm elements in E, a neighbor search is performed which is possible in $O(\log nm)$ [24, 27]. For each of the potential nm neighbors (*e.g.*, when all elements are at the same point) three constant runtime operations are performed in Line 12–15. This results in a complexity of $O(nm \cdot (\log nm + nm \cdot 1)) = O(n^2m^2)$.

***Matching*:** The matching phase operates on the set of possible pairs $\hat{P}$ to match. In the worst case, all elements from other models are valid match candidates for an element e during Phase 2. Thus, $|\hat{P}| \leq (nm)^2$ and sorting $\hat{P}$ in Line 18 requires $O(n^2m^2 \log nm)$ steps in the worst case. Constructing T in Line 19 is possible in $O(nm)$. The steps inside the loop at Line 20 have to be repeated $O(n^2m^2)$ times because $|\hat{P}| \leq (nm)^2$. Searching for matches t, t' in Line 21 and 22 has a worst case complexity of $O(nm)$ because $|T| \leq nm$. Merging the matches in Line 23 is $O(n)$ as valid matches only contain at most one element per model, *i.e.*, $|t \cup t'| \leq n$. For the same reason, *shouldMatch* in Line 24 requires $O(n)$ steps. Line 25 exhibits worst-case runtime of $O(nm)$. We get $O(n^2m^2 \log nm + n^2m^2 \cdot nm) = O(n^3m^3)$.

***Overall Complexity*:** The matching phase dominates the runtime complexity: We get $O(n^2m^2 + n^2m^2 + n^3m^3) = O(n^3m^3)$ in the worst case, which is an improvement over NwM's worst-case complexity of $\mathcal{O}(n^4m^4)$ [135]. In practice, we expect a much lower runtime complexity because Phase 1 and 2 of RaQuN are dedicated to reduce the number of comparisons in Phase 3, while the estimation of the worst case complexity assumes no reduction. It is highly unlikely that all elements are mapped to the same point in the vector space such that all pairs of elements become potential match candidates in $\hat{P}$.

3.2.4 Configuration Options

In this section, we discuss the variation points of RaQuN. For each of them, we propose a domain-agnostic configuration option such that RaQuN can be applied to models of any type. In the following, we discuss possible adjustments and implementations for the different variation points in each phase of RaQuN.

Candidate Initialization

The candidate initialization has two points of variation: the *multi-dimensional search tree* and the *vectorization*.

RaQuN can construct the vector space with any multi-dimensional data structure supporting *insertion* and *neighbor search*, such as kd-trees[24].

The vectorization function defines the abstraction of model elements and their properties. It embodies RaQuN's core trade-off between runtime performance and matching quality, as it directly impacts which match candidates are retrieved and the computational effort of retrieval. Generally speaking, a vectorization function should cluster similar elements in the same region of the vector space. If more dimensions are used for vectorization, the level of abstraction is lower and the clustering of similar elements is improved, but the nearest neighbor search on the tree requires more time. If less dimensions are used, the level of abstraction is greater, which can reduce the time required to find match candidates significantly, but it also becomes less likely that suitable match candidates can be found among the neighbors in the vector space. This could negatively affect the quality of the matching, as more incorrect or missing matches might be produced.

In the following, we discuss two examples of possible vectorization functions: a low-dimensional vectorization (Low Dim) and a high-dimensional vectorization (High Dim). These are two concrete suggestions which can be applied to any element/property representation of a model; Low Dim is in favor of performance and High Dim is in favor of matching quality.

***Low Dim*:** The low-dimensional vectorization reuses the two dimensions of the very simple vectorization presented in Section 3.2.3. These two dimensions encode the average number of characters in an element's properties, and its total number of properties. Additionally, for each unique character in an element's properties, there is one dimension that represents the number of occurrences of that character in the element's properties.The number of dimensions is bound by the size of the alphabet, and may be reduced by omitting those dimensions which represent characters that do not occur in any property name.

***High Dim*:** The high-dimensional vectorization represents all distinct properties of model elements of all input models by a dedicated dimension of the vector space $\{0, 1\}^K$, where K is the number of distinct properties in all elements. An element is represented by a bit vector in this space; the value at the index representing a dedicated property is set to 1 if the element has that property, and 0 otherwise. The number of required dimensions dynamically grows with the number of distinct properties, and thus with number and size of input models.

Candidate Search

The candidate search is configured by the *number of considered nearest neighbors* k' and the *distance metric*.

The parameter k' determines how many neighbors are retrieved for each element, which directly influences how many candidate pairs p are considered during the matching phase. Increasing k' leads to more candidate pairs. Each neighbor will be less significant than the previous one as nearer (more similar) neighbors are considered first. While an optimal value of k' can only be determined empirically with respect to a dedicated measure of matching quality, a reasonable starting point for this is to set $k'=n$, as in our illustration in Section 3.2.3. The rationale behind this is that each element may have at most one corresponding element per input model, limiting the number of corresponding elements to n-1. The choice of n respects that the nearest neighbor search considers the query point itself as first neighbor.

The distance metric is used to determine the distance between the vector representations of two elements in the vector space. The metric influences which elements are considered close or distant to each other (*i.e.*, which elements are considered to be neighbors). In this work, we use the Euclidean distance, leaving experimentation with other distance metrics such as Cosine similarity or any custom metric (*e.g.*, a metric emphasizing specific dimensions) for future work.

Candidate Matching

In RaQuN's final matching phase, potential match candidates are compared directly according to their *similarity*, and the *shouldMatch* predicate determines whether candidates should be formed to actual matches. The purpose of *shouldMatch* is to compensate potential inaccuracy from abstracting elements by numerical vectors. In general, an implementation of *shouldMatch* could work on concrete model representations, consider meta-data related to the models, etc. To stay independent of such domain-specific aspects, in this work, we stick to relying only on the generic element-property representations when implementing *shouldMatch*.

The *similarity* function, which determines the similarity of elements, is applied to assess the quality of a matching as illustrated in Section 3.2.2. It is used to sort the match candidates $\hat{P}$ in Line 18 such that more similar pairs are considered to be merged first. In the following, we discuss two possible similarity functions and their corresponding *shouldMatch* predicate.

Weight Metric: The first similarity function is a *weight* metric by Rubin and Chechik [135], which assigns weight $w(t) \in [0, 1]$ to a match depending on the number of common properties and the number of elements in the match. Given match t, the weight is calculated as

$$w(t) = \frac{\sum_{2 \leq j \leq |t|} j^2 \cdot n_j^p}{n^2 \cdot |\pi(t)|} \tag{3.2}$$

where $|t|$ denotes the size of the match, n_j^p the number of properties that occur in exactly j elements of the match, and $\pi(t)$ is the set of all distinct properties of all elements in t.

For the configuration of *shouldMatch*[15] we follow the match decision proposed by Rubin and Chechik [135]. The idea is that any extension of a match should increase the quality of the overall matching. Two matches t and t' are merged if the weight of the merged match $t \cup t'$ is greater than the sum of the individual match weights:

$$shouldMatch_w(t, t', e, e') := w(t \cup t') > w(t) + w(t') \tag{3.3}$$

Jaccard Index: The second similarity function is the Jaccard Index, which we applied in our motivating example in Section 3.2.3. The Jaccard Index is a wide-spread similarity metric for sets; it is named after Paul Jaccard who first defined it as *coefficient de communauté* in his work on flora in the alpine zone [20]. The Jaccard Index can be applied to our matching problem, because elements are sets of properties. We want to match elements that have many common properties, while having only few individual properties. Given a match t, the Jaccard index is calculated as

$$J(t) = \frac{\bigcap_{e \in t} e}{\bigcup_{e \in t} e} \tag{3.4}$$

where e is an element, which we consider to be a set of properties. A greater Jaccard Index index corresponds to greater similarity.

Regarding *shouldMatch*, we define that elements should be matched if their similarity is greater or equal to a predefined similarity threshold s:

$$shouldMatch_J(t, t', e, e') := J(t \cup t') \geq s \tag{3.5}$$

While *shouldMatch* used by the weight metric matches two elements greedily (*i.e.*, elements can be matched if they have at least one common property), a threshold-based definition allows the specification of the desired minimum similarity of matches. Thereby, it is also possible that extending a match might decrease its match quality, as long as the minimum similarity is kept. It is not feasible to use a similarity threshold for the weight metric, because the weight also depends on the size of the match and the number of considered models; different thresholds would have to be applied depending on the considered models and the current match size.

We expect that each configuration option presented in this section has an impact on RaQuN's runtime and matching quality. We want to investigate this impact empirically.

3.2.5 Evaluation

In addition to our conceptual and theoretical contributions, we conduct an empirical investigation on a variety of datasets. We are interested in whether RaQuN scales for large models while achieving high matching quality. The full reproduction package can be found on Zenodo [336] and GitHub[16].

[15] The *shouldMatch* predicate takes the two matches, t and t', and the elements in the candidate pair, e and e', as input and returns $true$ or $false$. While e and e' are not used by the *shouldMatch* predicates presented here, we extended the predicate's interface to allow for match decisions that explicitly take e and e' into account.

[16] `https://github.com/AlexanderSchultheiss/RaQuN` (visited on 15/12/2025)

Table 3.4. Selected algorithms and their configurations.

Name	Type	Vectorization F.	Similarity F.
RaQuN Low Dim	n-way	Low Dim	Weight
RaQuN High Dim	n-way	High Dim	Weight
RaQuN Jaccard	n-way	Low Dim	Jaccard Index
NwM	n-way	N/A	Weight
Pairwise Ascending	two-way	N/A	Weight
Pairwise Descending	two-way	N/A	Weight

RQ1 How does the configuration of RaQuN's candidate initialization (*i.e.*, vectorization) affect its matching quality and runtime?

RQ2 Is $k' = n$ a suitable heuristic for the number of considered neighbors during RaQuN's candidate search?

RQ3 How does the configuration of RaQuN's candidate matching affect its matching quality?

RQ4 How does RaQuN perform compared to NwM and sequential two-way matching in terms of matching quality and runtime?

RQ5 How does RaQuN scale with growing model sizes?

Selected Algorithms

Table 3.4 summarizes the matchers we used for our experiments; all matchers are implemented in Java. We compare different configurations of RaQuN, and RaQuN with NwM and two sequential two-way approaches (Pairwise).

Prototypical Implementation of RaQuN: We implemented a prototype of RaQuN which uses a generic kd-tree implemented by the Savarese Software Research Corporation [229] as multi-dimensional search tree. For all other variation points, we implemented the domain-agnostic configuration options discussed in subsection 3.2.4 as extension of the prototype. First, for the comparison of vectorization functions (see Section 3.2.4), we implemented RaQuN Low Dim and RaQuN High Dim named according to their vectorization function; both use the weight metric as similarity function. Second, for the comparison of similarity functions (see Section 3.2.4), we additionally implemented RaQuN Jaccard using the Low Dim vectorization function and the Jaccard Index as similarity function.

Baseline Algorithms: All baseline matchers use the weight metric [135], defined in Equation (3.2), as similarity function (see Section 3.2.2). The prevalent way to calculate n-way matchings is sequential two-way [118, 130, 168, 202, 203, 305]. This leaves open (a) which two-way matching algorithm is used in each iteration, and (b) the order in which

Table 3.5. Experimental subjects and their characteristics.

			Elements		Properties	
	Model Type	#Models	Avg.	Median	Avg.	Median
Hospital	Simple class diag.	8	27.62	26	4.84	4
Warehouse	Simple class diag.	16	24.25	22	3.65	3
Random	Synthetic	100	26.99	26	5.36	5
Loose	Synthetic	100	28.88	29	4.43	4
Tight	Synthetic	100	25.01	25	8.79	9
Apo-Games	Simple class diag.	20	63.05	60	19.62	13
PPU Structure	SysML block diag.	13	32.15	32	3.26	2
PPU Behavior	UML statemachines	13	221.85	228	5.04	5
bCMS	UML class diag.	14	67.71	63	3.60	2
BCS	Component/connector	18	78.78	72	5.81	4
ArgoUML	UML class diag.	7	1,752.86	1,749	9.05	4
DAS	Simulink	19	842.37	879	11.00	11
APS	Simulink	7	202.71	206	11.00	11
APS-TL	Simulink	5	181.20	165	11.00	11
MRC	Simulink	3	773.33	970	11.00	11
WEC	Simulink	6	650.00	668	11.00	11

inputs are processed. For (a) we use the Hungarian algorithm [22] to maximize the weight of the matching in each iteration. For (b), Rubin and Chechik [135] report the most promising results for the *Ascending* and *Descending* strategies, which sort the input models by number of elements in ascending and descending order, respectively. For NwM, we use the prototype implementation provided by Rubin and Chechik [135].

Experimental Subjects

Our experimental subjects and their basic characteristics are summarized in Table 3.5. Converting the experimental subjects into element/property representations requires a pre-processing step that is model-type and technology-specific. We used the generic EMF model traversal and reflective API to access an element's local properties and referenced elements. Elements and properties of the Simulink models were accessed via basic Simulink getter routines, as well. Our pre-processing code is part of our reproduction package [336].

Experimental Subjects of Rubin and Chechik: To enable a fair comparison with NwM, the first five subjects selected for our evaluation stem from the n-way model matching benchmark set used by Rubin and Chechik [135]. The *Hospital* and *Warehouse* datasets include sets of student-built requirements models of a medical information and a digital warehouse management system, for both of which variation arises from taking different viewpoints. Both datasets originate from case studies conducted in a Master's thesis by Rad and Jabbari [115]. The latter three datasets have been synthetically created using a model generator, which in the *Random* case mimics the characteristics of the hospital and warehouse models. The *Loose* scenario exposes a larger range of model sizes and a smaller number of properties shared among the models' elements, while the *Tight* scenario exposes a smaller range w.r.t. these parameters.

Variants Generated from Product Lines: The second set of selected subjects are variant sets generated from model-based software product lines. We use a superset of the n-way model merging benchmark set used in a recent work of Reuling *et al.* [281].

The Pick and Place Unit (*PPU*) is a laboratory plant from the domain of industrial automation systems [166, 182] whose system structure and behavior are described in terms of SysML block diagrams and UML statemachines, respectively [158]. Variation arises from different scenarios supported by the plant. The Barbados Car Crash Crisis Management System (*bCMS*) [95, 96] supports the distributed crisis management by police and fire personnel for accidents on public roadways. We focus on the object-oriented implementation models of the system [95], including both functional and non-functional variability. The Body Comfort System (*BCS*) [112] is a case study from the automotive domain whose software can be configured w.r.t. the physical setup of electronic control units. We use the component/connector models of BCS, specifying the software architecture of the 18 variants sampled by Lity *et al.* [112]. *ArgoUML* is a publicly available CASE-tool supporting model-driven engineering with the UML. It was used in prior studies [150, 245] and provides a ground truth for assessing the quality of a matching using precision and recall. The dataset comprises detailed class models of the Java implementation [220]. They represent different tool variants which have been extracted by removing specific features for supporting different UML diagrams.

Variant Sets Created Through Clone-and-Own: Another subject stems from a software family called *Apo-Games* which has been developed using the clone-and-own approach [128, 136] (*i.e.*, new variants were created by copying and adapting an existing one) and which has been recently presented as a challenge for variability mining [243]. The challenge comprises 20 Java and five Android variants, from which we selected the Java variants only.

Simulink Subjects: We also included five Simulink subjects. Three of them are case studies taken from Schlie *et al.* [304, 322]: DAS, a driver assistance system from the SPES_XT project [338], and APS, and APS_TL, an auto platooning system from the CrEst project [375]. Schlie *et al.* extracted *module building blocks* from the Simulink models, which he then recombined in different combinations to generate variants of the three systems. We followed his process to generate 19, 7, and 5 variance models, respectively.

As Boll *et al.* previously found open source Simulink models to be suitable for empirical research [1], we mined GitHub for open-source projects with Simulink models. We found 317 distinct projects with 4,402 Simulink models. In this set, we conducted a basic search for Simulink model "twins", by looking for models with identical qualified names, *i.e.*, having the same subdirectory path and file name. Our intention of this was finding variations of Simulink models in different forks. We view these forked Simulink models as a substitute for variants. To this end, we investigated the "twins" and rejected identical models (by hashsum and then manual inspection), models of trivial size, and models without any matchable

Table 3.6. ArgoUML subsets and their characteristics.

		Elements		Properties	
	Size	Avg.	Median	Avg.	Median
Argo-Subset-1	1%	18.52	19	8.88	4
Argo-Subset-5	5%	93.83	94	8.61	4
Argo-Subset-10	10%	187.16	187	8.57	4
Argo-Subset-15	15%	278.39	278	8.51	4
Argo-Subset-20	20%	369.08	369	8.61	4
Argo-Subset-25	25%	459.68	460	8.59	4
Argo-Subset-30	30%	549.15	549	8.53	4
Argo-Subset-35	35%	637.57	638	8.58	4
Argo-Subset-40	40%	726.04	725	8.49	4
Argo-Subset-45	45%	813.00	813	8.54	4
Argo-Subset-50	50%	900.15	899	8.54	4
Argo-Subset-55	55%	987.39	987	8.60	4
Argo-Subset-60	60%	1,073.47	1,073	8.59	4
Argo-Subset-65	65%	1,159.51	1,159	8.60	4
Argo-Subset-70	70%	1,245.00	1,244	8.61	4
Argo-Subset-75	75%	1,330.52	1,329	8.61	4
Argo-Subset-80	80%	1,415.41	1,416	8.64	4
Argo-Subset-85	85%	1,499.65	1,496	8.65	4
Argo-Subset-90	90%	1,584.42	1,583	8.67	4
Argo-Subset-95	95%	1,668.24	1,665	8.68	4
ArgoUML	100%	1,752.86	1,749	9.05	4

elements. This search and filtering yielded two families: MRC comprising three[17] and WEC comprising six Simulink models.[18]

Generation of ArgoUML Subsets: As already mentioned, realistic applications of n-way matching in practice typically have to deal with large models but only a few model variants. Thus, we are primarily interested in how the algorithms scale with growing model sizes for a fixed number of model variants. Answering this question requires experimental subjects with a stepwise size increase.

To that end, in addition to the presented experimental subjects, we generated subsets of ArgoUML, which comprises the largest models of our subjects. The subjects are presented in Table 3.6; all subsets have the same number of models as ArgoUML but vary in the number of elements. The number of elements in each subset is a fixed percentage between 5% and 100% of the number of elements in ArgoUML. We increased the percentages in 5% steps and generated 30 subsets for each percentage, in addition to 30 subsets with 1% of elements.

The sub-models are generated as follows. First, we randomly select a subset of classes from the set of all classes of a given model such that the subset contains the desired percentage

[17] These stem from the MRC contest `https://de.mathworks.com/matlabcentral/fileexchange/50227-mission-on-mars-robot-challenge-2015-france`. (visited on 15/12/2025) Contestants constructed variants of a robot that identifies obstacles and avoids them.

[18] Here, forks modified a library model stemming from the open-source wave energy conversion simulator (WEC) `https://wec-sim.github.io/WEC-Sim/master/index.html`. (visited on 15/12/2025)

of the overall number of classes. We repeat this for each model in ArgoUML so that the number of models remains the same. Second, we eliminate properties corresponding to dangling references in the selected classes, such that no typed property references a class which is not contained in the subset of selected classes.

Evaluation Metrics

While measuring efficiency is a largely straightforward micro benchmarking task, there exists no generally accepted definition of the quality of a matching in the literature [281]. We use the two most widely established quality evaluation metrics *weight* and *precision/recall.*

Weight: One way to measure the quality of an n-way matching is the weight metric [135], which we also use as a similarity function (see Section 3.2.4), where the optimal matching is the one with the highest weight, expressed as the sum of the individual match weights. Given a matching T, its weight is calculated as $w(T) = \sum_{t \in T} w(t)$, where $w(t)$ is calculated as in Equation 3.2. There can be several matchings with the same weight, and thus several optimal matchings for a set of models. We chose the weight metric as it does not depend on a ground truth, which is often not available.

Precision/Recall: In the context of two-way matching, the quality of a matching is often assessed using oracles and traditional measures (*i.e.*, precision and recall) known from the field of information retrieval [40]. For our experimental subjects, however, such oracles are only available for models generated from a software product line. Here, unique identifiers $id(e)$ may be attached to all model elements e of the integrated code base and serve as oracles when being preserved by the model generation. This way, corresponding elements have the same ID. These IDs are generally not available for models that did not originate from a product line (*e.g.*, models created through cloning), and they are not exploited by the matching algorithms used in our experiments.

Each two-element subset of a valid match is considered a true positive TP if its elements share the same ID. If these elements have different IDs, they are considered false positive FP. Two elements sharing the same ID but being in distinct matches are considered false negatives FN. The amount of TP, FP, and FN is defined over all the matches in T:

$$TP(T) = \sum_{t \in T} \left| \Big\{ \{e_1, e_2\} \subseteq t \mid id(e_1) = id(e_2) \Big\} \right| \tag{3.6}$$

$$FP(T) = \sum_{t \in T} \left| \Big\{ \{e_1, e_2\} \subseteq t \mid id(e_1) \neq id(e_2) \Big\} \right| \tag{3.7}$$

$$FN(T) = \left| \bigcup_{\substack{t_1, t_2 \in T \\ t_1 \neq t_2}} \Big\{ \{e_1, e_2\} \mid e_1 \in t_1, e_2 \in t_2, \\ id(e_1) = id(e_2) \Big\} \right| \tag{3.8}$$

Precision, recall, and F-measure are calculated as usual [40]:

$$\text{precision(T)} = \frac{|TP(T)|}{|TP(T)| + |FP(T)|}$$

$$\text{recall(T)} = \frac{|TP(T)|}{|TP(T)| + |FN(T)|}$$

$$\text{F−measure(T)} = 2 \cdot \frac{precision(T) \cdot recall(T)}{precision(T) + recall(T)}$$

Precision expresses how many matches are correct, recall expresses how many required matches have been found, and F-measure is the harmonic mean of precision and recall.

Methodology and Results
We ran our experiments on a workstation with an Intel Xeon E7-4880 processor with a frequency of 2.90GHz. In order to reduce the influence of side-effects caused by additional workload on the experimental workstation, we run each algorithm 30 times on each of our experimental subjects, except for Random, Loose, and Tight for which we follow the methodology of Rubin and Chechik [135]. Here, we select 10 subsets comprising 10 of the 100 models for each run that is repeated 30 times, leading to 300 runs per algorithm and subject. Regardless of the experimental subject, we permutate the input models randomly for each experimental run to minimize the potential impact that the order of models might have on the result, due to RaQuN's *filterAndSort* (see Line 18) not determining a fixed order in the case of candidate pairs having equal similarity. We set a time-out of 12 hours for the matching of a dataset, due to the large amount of experimental runs.

RQ1: Configuration of the Candidate Initialization: To assess how the configuration of the candidate initialization (*i.e.*, which vectorization function is used) impacts RaQuN's performance, we compare RaQuN Low Dim and RaQuN High Dim on the experimental subjects presented in Section 3.2.5. Table 3.7 presents the average weight and runtime achieved by the two configurations. Here, we consider the weight metric as it does not require a ground truth, which is not available for all datasets.

With respect to runtime, both configurations can compute a matching for each of the experimental subjects, requiring at most a couple of minutes for all subjects besides ArgoUML. RaQuN Low Dim is significantly faster than RaQuN High Dim across all datasets; its smallest *relative* speed-up of a factor of 1.6 can be observed on DAS, and the largest relative speed-up of a factor of 71.0 on ArgoUML. In terms of *absolute* runtime differences, RaQuN Low Dim achieves only a minor advantage on small datasets (*e.g.*, Hospital, Warehouse, or Random), but it can compute a matching for ArgoUML – the experimental subject with the largest models – in less than one minute, while RaQuN High Dim requires almost an hour. With respect to weight, both configurations achieve similar matching weight on the majority of experimental subjects, but RaQuN High Dim computes matchings with higher weight on almost all experimental subjects.

These results show, that both of our generic vectorization functions lead to varying results, depending on the characteristics of the experimental subjects (see Table 3.5). This is

Table 3.7. Comparison of RaQuN configurations, averaged over 30 runs for each subject.

Algorithm	Hospital		Warehouse		Random		Loose	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**4.92**	0.08 [0.07, 0.12]	**1.63**	0.32 [0.27, 0.93]	**1.04**	0.07 [0.05, 0.13]	**1.03**	0.13 [0.07, 0.33]
RaQuN Low Dim.	4.04	**0.01** [0.01, 0.03]	1.53	**0.05** [0.04, 0.08]	0.82	**0.02** [0.01, 0.18]	0.77	**0.01** [0.01, 0.08]

Algorithm	Tight		Apo-Games		PPU Structure		PPU Behavior	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**0.94**	0.07 [0.05, 0.11]	**18.27**	71.12 [41.30, 104.58]	**28.95**	0.85 [0.81, 0.90]	**164.53**	18.32 [16.39, 20.34]
RaQuN Low Dim.	0.84	**0.02** [0.01, 0.10]	**18.27**	**1.23** [1.16, 1.40]	**28.95**	**0.10** [0.09, 0.17]	164.26	**9.61** [9.12, 10.90]

Algorithm	bCMS		BCS		ArgoUML		DAS	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**41.53**	2.84 [1.40, 3.58]	**51.17**	12.58 [8.10, 18.22]	**1727.65**	2,647.76 [1,483.48, 3,324.70]	732.27	1,073.55 [943.73, 1,272.92]
RaQuN Low Dim.	41.47	**0.22** [0.20, 0.32]	51.16	**5.75** [5.45, 6.72]	1727.45	**37.71** [33.56, 45.65]	**732.31**	**669.67** [644.30, 748.11]

Algorithm	APS		APS-TL		MRC		WEC	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**169.41**	3.74 [2.06, 7.26]	**153.72**	1.78 [0.78, 5.55]	**534.22**	51.39 [31.97, 94.79]	**263.14**	269.11 [201.61, 536.52]
RaQuN Low Dim.	**169.41**	**1.41** [1.32, 1.56]	**153.72**	**0.36** [0.33, 0.52]	530.16	**1.01** [0.87, 1.26]	257.97	**5.13** [3.85, 6.20]

not surprising, because mapping elements to points in the vector space based on property names (High Dim), or based on the characters used in the properties' names (Low Dim), is directly affected by the characteristics of the subject. The only subject, on which RaQuN High Dim computes a matching with slightly lower weight, is DAS. DAS is also the subject on which the smallest relative runtime difference was measured, which suggests that, for DAS, both vectorization functions lead to a similar mapping of elements to points in the vector space.

RQ1: *How does the configuration of RaQuN's candidate initialization (*i.e., *vectorization) affect its matching quality and runtime?*
Both generic configurations, RaQuN Low Dim and RaQuN High Dim, achieve their intended purpose and make it feasible to compute a matching for each experimental subject. The configuration of the candidate initialization has a significant impact on the runtime and match quality. While there is a noticeable trade-off between runtime and weight, opting for the maximization of weight is reasonable for almost all subjects.

RQ2: Suitability of k' Heuristic During Candidate Search: In order to assess the suitability of $k' = n$ as heuristic for the number of neighbors during the candidate search, we ran RaQuN Low Dim and RaQuN High Dim with increasing values of k'. We observed highly similar results for RaQuN Low Dim and RaQuN High Dim, and we thus discuss the results for RaQuN High Dim in the remainder of this section to reduce redundancy.

Figure 3.12 presents the results of the runs of RaQuN High Dim conducted on the datasets PPU, bCMS, and ArgoUML. The plots display the value of k' against the runtime of RaQuN.

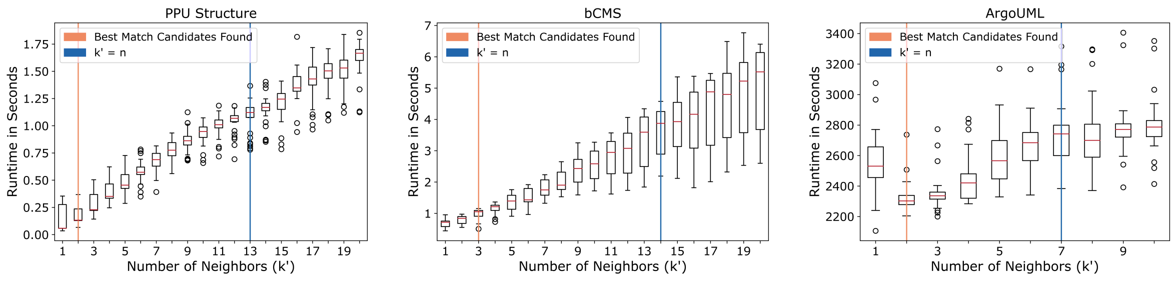


Figure 3.12. Impact of an increasing number of neighbors considered for matching on the performance of RaQuN High Dim.

The dark red line marks the k' at which the candidate search retrieved all match candidates necessary to reach the peak weight performance of RaQuN. The dark blue line marks the k' that is equal to the number of models n, which we propose as a possible heuristic for k'.

Our findings show that setting $k' = n$ made it possible to achieve the best matching possible with RaQuN. RaQuN was able to find the best candidates with small values of k', due to multiple elements being mapped to the same neighboring point in the vector space, and due to considering the neighbors for each element individually. Selecting a higher value for k' does not deteriorate the match quality, because the final match decision depends on *shouldMatch*. Moreover, the runtime of RaQuN shows a linear growth with higher k', which indicates that considering more neighbors than necessary will not cause a sudden increase in runtime.

Table 3.8 presents an overview of how many comparisons are saved by the candidate search (using k'=n). For most experimental subjects, RaQuN is able to reduce the number of comparisons by more than 90%. PPU is the only subject on which we achieve a rather low reduction of 48.5%. This is due to the high similarity between the elements, and the fact that the models are relatively small.

RQ2: *Is $k' = n$ a suitable heuristic for the number of considered neighbors during RaQuN's candidate search?*
With the heuristic choice of k'=n, RaQuN retrieves enough candidates for good matches, while still reducing the number of element comparisons by more than 90% for most experimental subjects.

RQ3: Configuration of RaQuN's Candidate Matching: RaQuN's third configuration aspect is the similarity function and its *shouldMatch* predicate (see Section 3.2.4). We compare the weight metric and the Jaccard Index as two possible options for the configuration of RaQuN's matching phase (see Section 3.2.4). For the Jaccard Index, we also have to set a value for the similarity threshold of its *shouldMatch* predicate (see Section 3.2.4). In practice, we envision that the similarity threshold is customized with respect to the similarity required for subsequent development activities. However, for the sake of evaluating RaQuN, we consider model matching independent of subsequent activities. Instead, we evaluate the Jaccard Index with a range of similarity thresholds from 0.25 through 1.00 in steps of 0.25.

Furthermore, while using the weight metric to assess the quality of matches is valid when comparing matchers that all rely on the same similarity function, it suffers from a bias when comparing different similarity functions. More specifically, we cannot conduct a comparison of matchers using the weight metric and matchers using the Jaccard Index as shown in Table 3.7, because using the weight metric for evaluation would favor matchers that internally use the weight metric to decide whether elements should be matched. Therefore, we answer the research question by matching our ArgoUML subsets. The subsets comprise unique identifiers making it possible to calculate precision, recall, and F-measure (see Section 3.2.5), which we consider to be unbiased evaluation metrics for the comparison of different similarity functions. Figure 3.13 presents the average precision, recall, and F-measure of RaQuN Low Dim using the weight metric, and RaQuN Jaccard using the Jaccard Index.[19]

First, when considering precision (*i.e.*, how many formed matches are correct) achieved by RaQuN Low Dim and RaQuN Jaccard, we observe that the precision of all matchers increases with increasing subset size. This is because the subsets are generated randomly by removing elements from the models. In turn, elements in the smaller subsets have fewer corresponding elements in other models, which increases the chance of matching elements that should not be matched. Furthermore, we observe differences between the precision of the matchers, depending on the similarity threshold of the Jaccard Index: Generally speaking,

[19] Both configurations use the Low Dim vectorization to reduce the runtime of the experiment.

Table 3.8. Number of element comparisons that are saved by RaQuN High Dim with k'=n.

Dataset	Full N-Way Matching	RaQuN	
	#Comparisons	#Comparisons	Saved
Hospital	21,211	936	95.6%
Warehouse	70,037	4,044	94.2%
Random	27,918	1,964	93.0%
Loose	26,716	1,995	92.5%
Tight	28,982	1,569	94.6%
Apo-Games	750,319	31,028	95.9%
PPU Structure	80,620	30,853	61.7%
PPU Behavior	3,814,644	207,385	94.6%
bCMS	416,571	44,336	89.4%
BCS	939,346	164,860	82.4%
ArgoUML	64,521,622	362,890	99.4%
DAS	121,115,254	3,700,634	96.9%
APS	858,294	33,500	96.1%
APS-TL	325,143	9,584	97.1%
MRC	1,675,724	6,745	99.6%
WEC	6,128,714	24,306	99.6%

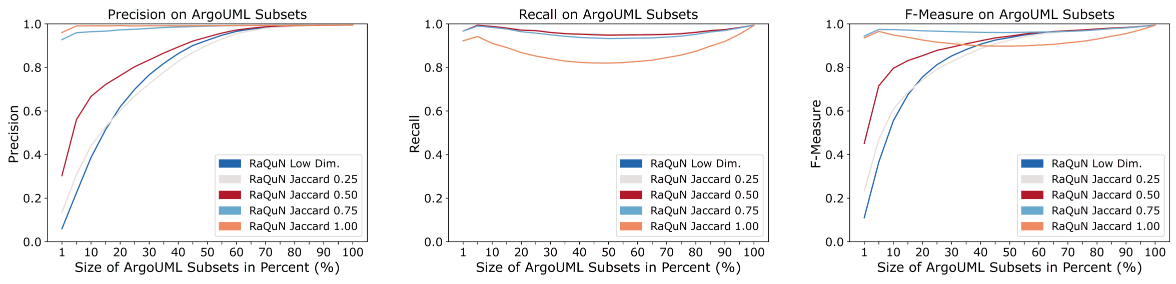


Figure 3.13. Average precision, recall and F-measure of RaQuN Low Dim and RaQuN Jaccard on subsets of ArgoUML with increasing size. RaQuN Jaccard was run with varying thresholds for its *shouldMatch* predicate, ranging from 0.25 to 1.00.

a higher threshold leads to higher precision. We expected this result because the likelihood of a match being correct correlates with the similarity of its elements. This is also the reason why RaQuN Low Dim achieves similar precision as RaQuN Jaccard with a threshold of 0.25; the weight metric forms matches greedily (*i.e.*, elements can be matched if they have at least one common property), which is comparable to a small similarity threshold.

Second, with respect to recall (*i.e.*, have all required matches been formed), we observe almost no difference between the two similarity functions. RaQuN Low Dim and RaQuN Jaccard achieve a high recall between 0.95 and 1.00 across all datasets. Only RaQuN Jaccard using a similarity threshold of 1.00 achieves a significantly lower recall across all subsets. This is not surprising, because a threshold of 1.00 only matches elements that have exactly the same set of properties, while a match can also be correct if the elements have a few different properties.

Finally, with respect to F-measure (*i.e.*, the harmonic mean between precision and recall), the results show that RaQuN Jaccard using a similarity threshold of 0.75 achieved the best matching quality across all subsets, and that RaQuN Low Dim and RaQuN Jaccard with thresholds of 0.25 and 1.00 achieved the worst overall matching quality depending on the subset size.

***RQ3:** How does the configuration of RaQuN's candidate matching affect its matching quality?*
RaQuN Jaccard achieves similar or better matching quality than RaQuN Low Dim, depending on the chosen similarity threshold. By considering a range of possible thresholds, we found that RaQuN Jaccard with a high threshold (*i.e.*, greater than 0.75) can achieve almost perfect precision, but that selecting a too high threshold (*i.e.*, 1.00) negatively affects the recall. In practice, RaQuN should apply a similarity function with a similarity threshold in order to reduce the number of incorrect matches.

RQ4: Comparison With Other Algorithms: For the comparison of RaQuN against the baseline matchers NwM, Pairwise Ascending, and Pairwise Descending, we assess the differences in average runtime and match weight on each experimental subject. We further

Table 3.9. Comparison of different matching algorithms across all subjects, averaged over 30 runs.

Algorithm	Hospital		Warehouse		Random		Loose	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**4.92**	**0.08** [0.07, 0.12]	**1.63**	**0.32** [0.27, 0.93]	**1.04**	**0.07** [0.05, 0.13]	**1.03**	**0.13** [0.07, 0.33]
NwM	4.49	20.64 [16.18, 24.22]	1.46	74.51 [29.53, 84.87]	0.80	24.00 [1.23, 76.02]	0.79	22.40 [1.12, 70.68]
Pairwise Ascending	4.49	0.31 [0.28, 0.44]	1.11	0.36 [0.33, 0.45]	0.79	0.21 [0.10, 0.31]	0.74	0.14 [0.08, 0.22]
Pairwise Descending	4.72	0.16 [0.14, 0.22]	1.27	0.36 [0.32, 0.53]	0.78	0.15 [0.10, 0.23]	0.74	0.14 [0.08, 0.25]

Algorithm	Tight		Apo-Games		PPU Structure		PPU Behavior	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**0.94**	**0.07** [0.05, 0.11]	**18.27**	71.12 [41.30, 104.58]	**28.95**	0.85 [0.81, 0.90]	**164.53**	18.32 [16.39, 20.34]
NwM	0.88	39.75 [1.63, 66.60]	17.91	5,462.91 [3,742.22, 6,597.83]	28.64	9.27 [7.84, 10.07]	146.35	4,616.30 [3,410.86, 5,919.65]
Pairwise Ascending	**0.94**	0.22 [0.16, 0.34]	12.96	**10.42** [9.32, 12.04]	28.65	0.27 [0.22, 0.38]	145.46	11.03 [10.32, 12.61]
Pairwise Descending	0.93	0.22 [0.17, 0.33]	16.40	10.68 [9.27, 13.50]	28.89	**0.26** [0.21, 0.42]	142.56	**10.84** [10.20, 13.01]

Algorithm	bCMS		BCS		ArgoUML		DAS	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**41.53**	2.84 [1.40, 3.58]	**51.17**	12.58 [8.10, 18.22]	**1727.65**	2,647.76 [1,483.48, 3,324.70]	**732.27**	1,073.55 [943.73, 1,272.92]
NwM	41.25	247.31 [227.52, 275.95]	43.20	330.25 [235.74, 388.45]	- - - timeout - - -		- - - timeout - - -	
Pairwise Ascending	39.76	1.16 [1.13, 1.21]	43.83	5.69 [3.85, 7.43]	1702.91	318.26 [297.43, 379.76]	683.75	953.61 [774.91, 1,712.88]
Pairwise Descending	38.76	**1.04** [1.01, 1.07]	47.16	**4.84** [3.37, 6.73]	1710.25	**314.88** [302.28, 329.35]	730.40	**809.85** [757.50, 1,213.46]

Algorithm	APS		APS-TL		MRC		WEC	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**169.41**	**3.74** [2.06, 7.26]	**153.72**	**1.78** [0.78, 5.55]	**534.22**	51.39 [31.97, 94.79]	**263.14**	269.11 [201.61, 536.52]
NwM	- - - timeout - - -		- - - timeout - - -		- - - timeout - - -		- - - timeout - - -	
Pairwise Ascending	148.62	7.46 [4.73, 11.66]	136.02	3.69 [1.91, 9.16]	532.76	23.97 [13.63, 37.63]	253.47	65.37 [39.25, 110.43]
Pairwise Descending	165.28	4.93 [4.78, 5.80]	152.23	3.12 [1.99, 4.92]	534.18	**14.97** [13.70, 17.34]	258.05	**46.92** [39.91, 73.30]

evaluate the quality of matchings in terms of precision, recall, and F-measure on ArgoUML subsets. Based on our earlier conclusion that using the high-dimensional vectorization is the preferable choice (see Section 3.2.5), we consider RaQuN High Dim as representative of RaQuN. RaQuN High Dim uses the weight metric as similarity function, because the baseline matchers also use the weight metric.

Table 3.9 presents the average weight and runtime achieved by the matchers. RaQuN and Pairwise are significantly faster than NwM. While, on average, NwM requires between 9s and 75s for the matching of smaller subjects (< 50 elements) PPU Structure and Hospital through Tight, the other algorithms are able to calculate a matching in less than a second. Matchings for bCMS and BCS were calculated by RaQuN and Pairwise in less than 13s, where NwM required 247s and 330s. Moreover, NwM was not able to provide a matching for ArgoUML, DAS, APS, APS-TL, MRC, and WEC before reaching the time-out of 12h, and it took about 90min and 70min for processing Apo-Games and PPU Behavior, respectively. In contrast, RaQuN provides a matching in an average time of less than 45 minutes for ArgoUML, less than 20 minutes for DAS, 71s for Apo-Games, and less than 5 minutes for the remaining subjects.

When considering the achieved weights, RaQuN delivers the matchings with the highest weights for all datasets. NwM delivers higher weights than Pairwise for six of the eleven datasets. Notably, ascending and descending Pairwise always yield different weights, which confirms the observation by Rubin and Chechik that performance of sequential two-way

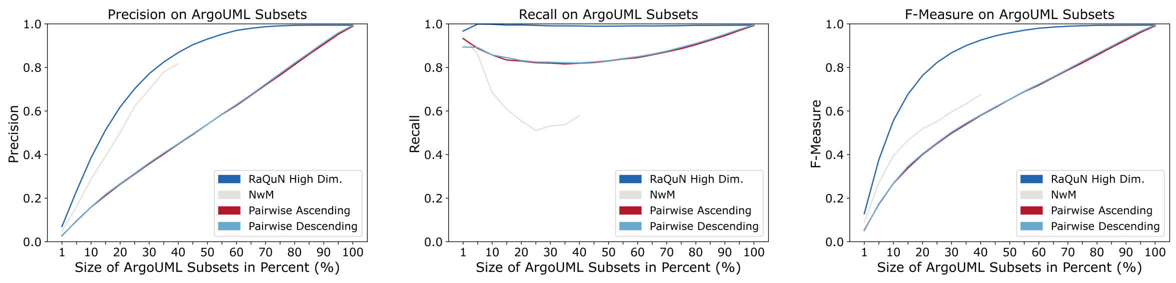


Figure 3.14. Average precision, recall and runtime of RaQuN, NwM, and Pairwise on subsets of ArgoUML with increasing size.

matchers depends on the order of input models [135]. Moreover, which of the two Pairwise matchers performs better changes from subject to subject, making it not possible to anticipate which order will yield better results.

The comparison of matching quality in terms of precision, recall, and F-measure is presented in Figure 3.14.

First, the precision achieved by the different algorithms is presented in the left-most plot of Figure 3.14; for NwM, we only have results for subsets with a size of up to 40%, because the timeout of 12 hours was reached for larger subsets. On the subset with only 1% of elements, the matching precision of all approaches lies at roughly 0.1. With increasing subset size, we note a significant difference in precision when we compare the n-way and sequential two-way approaches. Moreover, we can observe a slightly higher precision for RaQuN in comparison to NwM. The n-way algorithms deliver more precise matchings because Pairwise does not consider all possible match candidates for an element at once and therefore may form worse matches.

Second, the central plot of Figure 3.14 shows the recall achieved by the algorithms. For all algorithms, the recall first drops with increasing subset size and then rises again after reaching a subset size between 30% and 50%, depending on the algorithm. The reason for this is that, according to our ArgoUML subset generation, the number of elements initially grows faster than the number of properties of each element. The latter depends on the occurrence of other types in the model which may not be included in the sub-model yet. As a consequence, some matches are missed. While this effect is only barely noticeable for RaQuN, it is prominent for NwM and Pairwise. The comparably high recall achieved by RaQuN indicates that the vectorization is able to mitigate this effect. On the other hand, Pairwise shows a larger drop in recall, as forming incorrect matches (see precision) can additionally impair its ability to find all correct matches. To our surprise, the recall of NwM drops significantly more than the recall of Pairwise. We assume that the optimization step of NwM, which may split already formed matches into smaller ones, accounts for a higher loss in recall.

Lastly, the right-most plot of Figure 3.14 presents the F-measure achieved by the matchers. Here, we observe that RaQuN offers the best trade-off between precision and recall across all ArgoUML subsets.

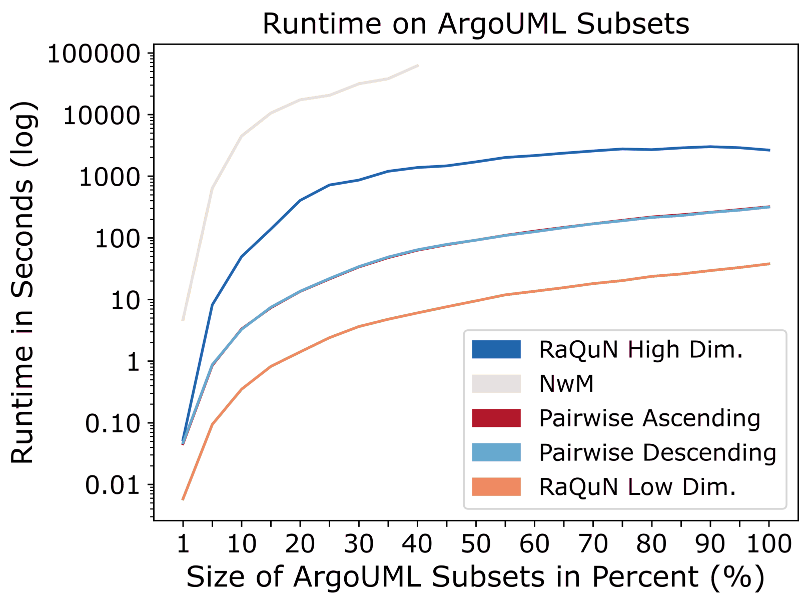


Figure 3.15. Average runtime of RaQuN, NwM, and Pairwise on subsets of ArgoUML with increasing size.

RQ4: *How does RaQuN perform compared to NwM and sequential two-way matching in terms of matching quality and runtime?*
RaQuN High Dim is significantly faster than NwM on all datasets, and is almost as fast as two-way matchers on medium-sized datasets with hundreds of elements. RaQuN High Dim achieves the highest weights across all experimental subjects and is able to deliver matchings with the highest precision and recall on the ArgoUML subsets.

RQ5: Scalability with Growing Input Size: The results of our scalability analysis on the ArgoUML subsets is shown in Figure 3.15, which presents the average logarithmic runtimes of the algorithms for each subset size. We observe that the runtime of NwM increases rapidly with the subset size. NwM requires more than 60 minutes on average to compute a matching on the 15% subsets. This confirms that it is not feasible to match larger models with NwM. In contrast, it is still feasible to run RaQuN and Pairwise on the full ArgoUML models. RaQuN and Pairwise show similar scaling properties, while RaQuN's absolute runtime depends on the used vectorization function. For matching the full models (see Table 3.9), the average runtime of RaQuN High Dim was less than 45 minutes, making it slower than Pairwise, but still feasible. On the other hand, the average runtime of RaQuN Low Dim was less than one minute making it significantly faster than even the Pairwise matchers.

RQ5: *How does RaQuN scale with growing model sizes?*
As opposed to NwM, RaQuN shows great scaling properties for models of increasing size, up to the largest models of our experimental subjects containing more than 10,000 elements in total. By using a vectorization function that opts for better runtime, RaQuN Low Dim computes matches even faster than Pairwise. This is a strong indicator that RaQuN's typical scaling behavior is considerably better than its theoretical worst case complexity.

Threats to Validity

Our experiments rely on evaluation metrics that may affect the *construct validity* of the results. First, we use the weight metric which has already been used in prior studies [135]. While it can be applied to compare the results on the same subject, weights obtained for different experimental subjects are hardly comparable. To that end, we use precision and recall [40] in order to assess the quality of the matchings for the ArgoUML subsets. The calculation of both depends on our definition of true positives, false positives, and false negatives. Here, we favored a pairwise comparison over a direct rating of complete matches to rate almost correct matches better than completely wrong matches. Another potential threat pertains the construction of ArgoUML subsets. Using unrelated models of different size would introduce the bias of varying characteristics of these models. Hence, we decided to remove parts of the largest available system. While we argue that this is the better choice, it is possible that the ArgoUML subsets do not represent realistic models. Moreover, using the product-line variants of ArgoUML, PPU, BCS, and bCMS as experimental subjects could have introduced a bias because these are derived from a clean and integrated code base, lacking unintentional divergence[131, 156, 176]. Thus, while it is common in the literature to use product-line datasets [150, 164, 174, 245] as they inherently provide ground-truth matchings, we also considered the clone-and-own system ApoGames.

Computational bias and random effects are a threat to the *internal validity*. Other processes on the machine may affect the runtime, but also the matching may differ in several runs with the same input. The non-determinism of RaQuN is due to the use of hash sets used in the implementation. Furthermore, the order in which matches are merged may vary for identical similarity scores. We mitigated those threats by repeating every measurement 30 times, each with a different permutation of the input models. Additionally, the random generation of the ArgoUML subsets might have introduced a bias favoring a particular algorithm. To mitigate this bias, we sampled 30 subsets for each subset size, totaling in 600 different subsets included in the reproduction package. Faults in the implementation may also affect the results. We implemented several unit tests for each class of RaQuN's implementation and manually tested the quality of RaQuN and the evaluation tools on smaller examples. Additionally, we resort to the original implementations of NwM and Pairwise.

The question whether the results generalize to other subjects, which must be converted to element/property models, is a threat to the *external validity*. We mitigate this threat by our selection of diverse experimental subjects. We used the experimental subjects from the original evaluation of NwM, for which Rubin and Chechik have already mitigated this threat [135]. Moreover, we have experimented with additional subjects covering (a) *different domains*, *i.e.*, information systems (bCMS), industrial plant automation (PPU), automotive software (BCS), software engineering tools (ArgoUML), and video games (Apo-Games), (b) *different origins*, *i.e.*, academic case studies on model-based software product lines (PPU, BCS, bCMS), a software product line which has been reverse engineered from a set of real-world software variants written in Java (ArgoUML), and a set of variants developed using clone-and-own (Apo-Games), and (c) *different model types*, *i.e.*, UML class diagrams (bCMS, ArgoUML), SysML block diagrams and UML statemachines (PPU), component/connector models (BCS), and Simulink models.

3.2.6 Related Work

Traditional matchers are two-way matchers which can be classified into *signature-based*, *similarity-based*, and *distance-based* approaches. Signature-based approaches match elements which are "identical" concerning their signature [70] - typically a hash value which comprises conceptual properties (*e.g.*, names) or surrogates (*e.g.*, persistent identifiers). Similarity-based matching algorithms try to match the most similar but not necessarily equal model elements [47, 53, 55, 57, 68, 73]. Distance-based approaches try to establish a matching which yields a minimal edit distance [31, 63, 66, 78, 123, 147]. Among these categories, signature-based matching is the only one which could be easily generalized to the n-way case. However, the limitations of signatures have been extensively discussed [47, 55, 72, 82, 111].

A few approaches realize n-way matching by the *repeated two-way matching* of the input artifacts [118, 130, 168, 202, 203, 305]. However, as reported by Rubin and Chechik [135] and now confirmed by our empirical evaluation, this may yield sub-optimal or even incorrect results as not all input artifacts are considered at the same time [135, 168].

To the best of our knowledge, Rubin and Chechik are the only ones who have studied the *simultaneous* matching of n input models [135]. Their algorithm called NwM applies iterative bipartite graph matching whose insufficient scalability motivated our research. RaQuN is radically different from NwM. It is the first algorithm applying index structures (*i.e.*, multi-dimensional search trees) to simultaneous n-way model matching (Phase 1 and Phase 2 in Algorithm 1). Even without these phases, the matching (Phase 3) differs from NwM by abstaining from bipartite graph-matching, reducing the worst-case complexity (see Section 3.2.3).

Our usage of multi-dimensional search trees is inspired by Treude *et al.* [72]. While they discuss basic ideas of how model elements can be mapped onto numerical vectors in the context of two-way matching, the actual matching problem was not even addressed but delegated to an existing two-way matcher. Moreover, a dedicated vectorization function needs to be provided for all types of model elements, while we work with a vectorization which is domain-agnostic.

All approaches to both n-way and two-way matching assume matches to be mutually disjoint and that no two elements of a match belong to the same input model. This is a reasonable assumption which we adopt in this paper to ensure the comparability of RaQuN with the state-of-the-art. The only exception which deviates from this assumption is the distance-based two-way approach presented by Kpodjedo *et al.* [132], which extends an approximate graph matching algorithm to handle many-to-many correspondences. Regarding the ground truth matchings of our experimental subjects obtained from product lines, there is no need for such an extension of n-way matching algorithms. However, it might be a valuable extension for some use cases (*e.g.*, for comparing models at different levels of abstraction) which we leave for future work.

Several approaches which can be characterized as *merge refactoring* have been proposed in the context of migrating a set of variants into an integrated software product line. Starting from a set of "anchor points" which indicate corresponding elements, the key idea is to extract the common parts in a step-wise manner through a series of variant-preserving refactorings [84, 93, 117, 136, 164, 200, 215, 279, 280, 281]. Anchor points may be determined through clone detection [84, 136, 200, 215] or conventional matchers [93, 117, 279, 280, 281], and may be corrected and improved by the merge refactoring. However, such implicit calculations of optimized n-way matchings require extensive catalogues of language-specific refactoring operations which have to be specified manually [200, 215, 279, 281]. Merge refactoring approaches are complementary to our approach, because they require sufficiently accurate matchings as input to avoid prohibitive computational efforts during refactoring [281].

Another approach for managing cloned software variants has been presented by Linsbauer *et al.* [170, 245]. They use combinatorics of feature configurations to map features to parts of development artifacts, which implicitly establishes n-way matchings. Similarly, implicit n-way matchings are established through extracting product-line architectures as, *e.g.*, proposed by Assunção *et al.* [287]. However, the required additional information such as complete feature configurations is typically not available.

Finally, Babur *et al.* [184, 206] cluster models in model repositories for the sake of repository analytics. They translate models into a vector representation to reuse clustering distance measures. However, clustering is performed on the granularity level of entire models, while our candidate initialization clusters individual model elements. In fact, as shown by Wille et al. [256], both may be used complementary by first partitioning a set of model variants and then performing a fine-grained n-way matching on clusters of similar models.

3.2.7 Conclusion and Future Work

Model matching is a major requirement in many fields, including extractive software product-line engineering and multi-view integration. In this paper, we proposed RaQuN, a generic algorithm for simultaneous n-way model matching which scales for large models. We achieved this by indexing model elements in a multi-dimensional search tree which allows for efficient range queries to find the most suitable matching candidates. We are the first to provide a thorough investigation of n-way model matching on large-scale subjects (ArgoUML) and a real-world clone-and-own subject (Apo-Games). Compared to the state-of-the-art, RaQuN is an order of magnitude faster while producing matchings of better quality. RaQuN makes it possible to adopt simultaneous n-way matching in practical model-driven development, where models serve as primary development artifacts and may easily comprise hundreds or even thousands of elements.

Our roadmap for future work is threefold. First, we plan an in-depth investigation of RaQuN's potential for domain-specific optimizations. For example, RaQuN could be adjusted to specific requirements of different application scenarios and characteristics of different types of models. Second, RaQuN, Pairwise, and NwM only support matching one element of a model to at most one element of each other model (1-to-1). This might limit the possibility to find the correct matches in certain cases (*e.g.*, an element was split into several smaller elements). Therefore, from a more general point of view, we want to extend simultaneous n-way model matching to support n-to-m matches for which we believe that RaQuN serves as a promising basis to enter and explore this new aspect of n-way matching. Third, in accordance with the state-of-the-art, RaQuN forms mutually disjoint matches. Therefore, an element belongs to at most one match and no alternative matches for an element are computed. We plan on supporting scenarios in which several match proposals instead of a single exact match for a specific element are desired (*e.g.*, scenarios in which a user interactively selects the most suitable match for an element).

3.3 Case Study 2: Simulink bus usage in practice: an empirical study

↪ Thesis Roadmap

Authors: Tiago Amorim, Alexander Boll, Ferry Bachmann, Timo Kehrer, Andreas Vogelsang, Hartmut Pohlheim.
Published in: Journal of Object Technology.

Abstract: Matlab/Simulink is a graphical modeling environment that has become the de facto standard for the industrial model-based development of embedded systems. Practitioners employ different structuring mechanisms to manage Simulink models' growing size and complexity. One important architectural element is the so-called bus, which can combine multiple signals into composite ones, thus, reducing a model's visual complexity. However, when and how to effectively use buses is a non-trivial design problem with several trade-offs. To date, only little guidance exists, often applied in an ad-hoc and subjective manner, leading to suboptimal designs. Using an inductive-deductive research approach, we conducted an exploratory survey among Simulink practitioners and extracted bus usage information from a corpus comprising 433 open-source Simulink models. We elicited 22 hypotheses on bus usage advantages, disadvantages, and best practices from the data, whose validity was later tested through a confirmatory survey. Our findings serve as requirements for static analysis tools and pave the way toward guidelines on bus usage in Simulink.

3.3.1 Introduction

Over the last two decades, Matlab/Simulink (Simulink, for short) has become the *de facto* standard for the industrial model-based development of embedded systems in various domains (*e.g.*, automotive, avionics, industrial automation, medicine) [83, 181], with millions of users [316] and thousands of companies[20] employing it. While Simulink started as a tool mainly for modeling and simulating single controllers, today, it can describe large systems comprising thousands of blocks communicating over signals [1]. This increasing complexity has triggered research on properly structuring large-scale Simulink models by adopting

[20] https://enlyft.com/tech/products/simulink (visited on 15/12/2025)

well-known software design principles [127, 161, 167, 247, 294, 295]. Current studies revolve around hierarchical decomposition, encapsulation, and information hiding, aiming at classical quality attributes such as modularity, coupling, and cohesion.

The *subsystem* and the *bus* are Simulink's architectural elements that address some of the aforementioned properties. The former provides the capability for encapsulating blocks and other nested subsystems. The latter allows combining individual signals into composite signals to keep them organized and reduce visual clutter[21]. However, buses have gained considerably less attention in the literature than subsystems, despite their importance for large and complex models. For instance, in one of the largest datasets [342] of public Simulink projects (similar to those used in [1, 337]), we found models that do use buses have a median of 22 subsystems and 239 blocks. In comparison, models without buses feature three subsystems and 29 blocks in the median.

Additionally, guidelines and best practices on bus use are severely limited. The existing ones, namely the MathWorks Advisory Board (MAB) guidelines and the Motor Industry Software Reliability Association (MISRA) guidelines [296, 379], provide little guidance on bus usage, apart from naming advice, label position, and suggesting not to mix MUX and bus elements. However, more advanced bus usage principles are implicit, ad-hoc, and subjective. This knowledge gap might lead to suboptimal designs, making it harder to understand, maintain, and evolve a complex model [76, 109].

Regardless of the lack of guidelines, deciding when and how to use them in a Simulink model is not trivial. Classical indicators naturally arising from principles such as functional decomposition or modularization are hardly applicable. Signals may cross the subsystem hierarchy, and buses are primarily used for graphically encapsulating a set of signals rather than physically reducing the coupling of connected elements. Modelers are faced with additional trade-offs regarding bus usage, such as reducing visual complexity and ease of bulk element manipulation versus the loss of visual information and the risk of bundling logically unrelated signals. In addition, modelers have to organize buses (*i.e.*, ordering and nesting of signals and choosing start and endpoints) to avoid buses that are too "clunky" [314].

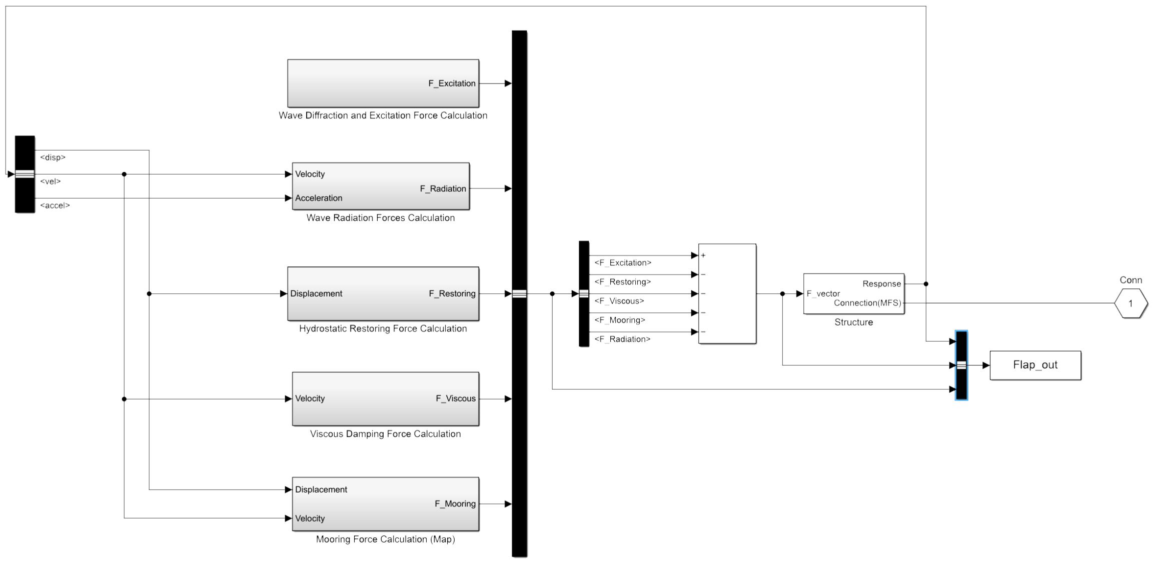
To better understand the trade-offs of using Simulink buses in practice, we devised an inductive-deductive research approach in a triangulation fashion [214]. We first performed an exploratory survey with Simulink practitioners and analyzed bus usage in 433 open-source Simulink models. From the gathered data, we elicited 22 hypotheses concerning the advantages of bus usage, bus size best practices, and situations when to use them and when to avoid them. We then conducted a confirmatory survey with Simulink practitioners, asking about their agreement with the hypotheses, thus testing their validity.

The majority of the hypotheses gained considerable support in the Confirmatory survey, making them candidates for being generally accepted. Thus, our findings provide an empirically grounded stepping stone that paves the way toward guidelines on Simulink bus usage. Next to supporting the design of large-scale Simulink models from a constructive perspective, such guidelines may further serve as requirements for static analysis tools.

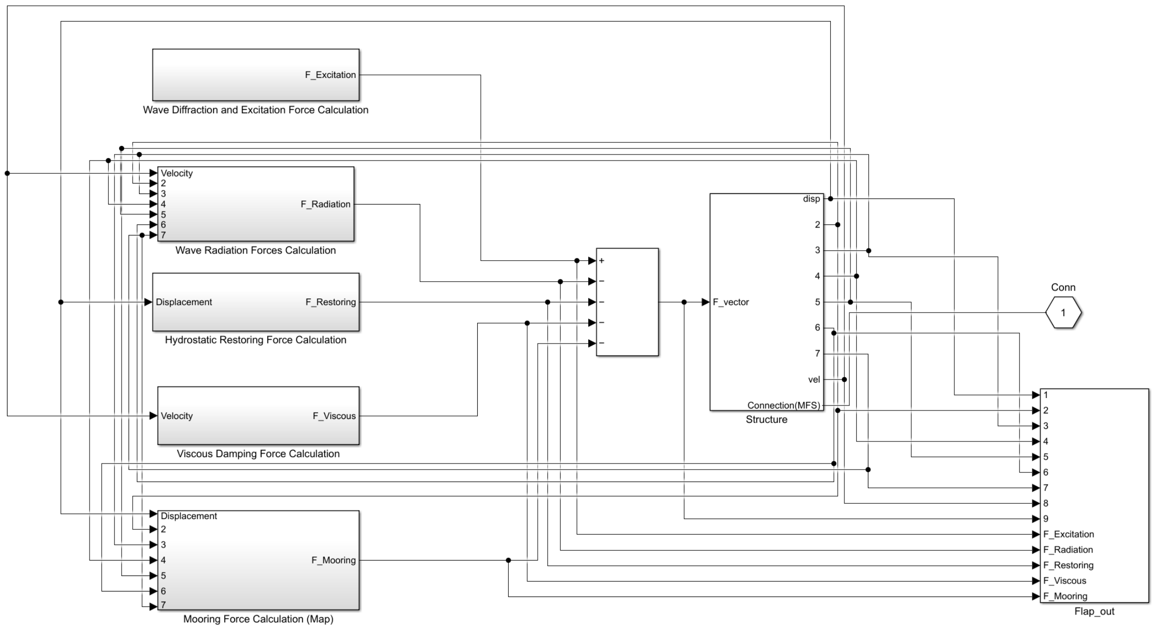
This study was conducted in the context of a research project[22] with an industrial partner,

[21] `https://www.mathworks.com/help/simulink/ug/composite-signal-techniques.html` (visited on 15/12/2025)

[22] SimuComp, 01IS18091 BMBF



(a) One view of a Simulink model, showing blocks connected by signal lines. The black blocks are bus creator blocks (single output on the right) and bus selector blocks (single input on the left). The white and shaded blocks are subsystems, which could be expanded to views of their currently hidden implementation details.



(b) An altered version of the model of Figure 3.16a, where we artificially resolved all buses and sub-buses up to the elementary signal lines. On the one hand, there are fewer blocks as the bus creators and selectors are removed. On the other hand, various new signal lines, block inputs, and outputs are now part of the diagram. Many of the signal lines cross other lines, and it is hard to discern which blocks are connected by which lines.

Figure 3.16. Two versions of an exemplary Simulink model showing blocks connected by signal lines. In Figure 3.16a bus creator and bus selector blocks encapsulate several signal lines. These signal lines can be seen once all buses are resolved in Figure 3.16b.

namely Model Engineering Solutions GmbH[23], which develops tools that support various Simulink model analyses. Our collaboration supports them in making an evidence-based and informed decision on whether they should extend their portfolio towards buses and how to extend their modeling guidelines and tool suite.

3.3.2 Background

Simulink is a graphical block-oriented modeling environment for simulating and analyzing multi-domain dynamical systems. Moreover, code generation facilities transform a model into a classical programming language, usually C. A Simulink model is a data flow graph whose vertices and edges are different kinds of *blocks* and *signal lines*, respectively. Blocks receive inputs from other blocks through signal lines connected by their *ports*. These inputs are processed and forwarded as outputs, yielding a data flow-oriented model [79]. Two versions of an exemplary model[24] are shown in Figure 3.16.

Simulink can display different views of a model and offers three means of abstraction and structuring a model: subsystems, MUXes, and buses. Their proper usage reduces the number of visible elements, which is achieved by encapsulating and hiding their contained elements from the outside. As a benefit, unnecessary details from the current view are hidden [172], thus, lowering the visual complexity. Further, these can give context information, making the elements' relationships more explicit. Finally, bulk operations are possible by encapsulating several elements, *i.e.*, copying or moving all encapsulated elements together. A subsystem can be compared to a function, method, or procedure of classical programming languages. The MUX can best be compared with an array and a bus with a struct or a dictionary [45]. In other words, elements of a MUX must be of the same type and are accessed by an index, while elements of a bus can have mixed types and are accessed by their name.

By default, a bus is only *virtual*, *i.e.*, it does not change a model's functionality if it were to be resolved into its elements. Therefore, it does not affect the simulation of a model or its code generation. However, engineers can change this characteristic to *non-virtual*, which forces the generation of C structs in the generated code, preserving the bus elements' cohesion. This way, all bus signals will be created in simulations, leading to decreased performance.

Buses are built by bundling signals in a block called the *bus creator*. This block receives multiple signal lines as inputs and a bus signal as output. To access a bus' elements, the *bus selector* block is used, where the single bus line enters and the user-selected line(s) exit(s). Both are depicted as black bars in Figure 3.16a. In Figure 3.16b, we resolved all buses of Figure 3.16a to show the effect bus usage can have.

On the one hand, without using buses, many new signal lines appear. It is difficult to say which blocks are connected due to the clustering of lines that cross each other at several intersection points. As the number of ports of each block also increases, some blocks need to be resized to carry all of them, highlighting another problem: the sheer number of ports makes it hard, if not impossible, to tell which port of which block is connected to which

[23] `https://model-engineers.com/en` (visited on 15/12/2025)

[24] We depict the subsystem WECSim_Lib/Body Elements/Rigid Body of the model source/lib/WEC-Sim/WEC_Sim_Lib.slx from the GitHub project `https://www.github.com/ratanakso/WEC-Sim-OSU-Temp` (visited on 15/12/2025)

Table 3.10. Overview of the demographic characteristics of the participants of our Exploratory survey.

ID	Application Domain	Role in Organization	Organization size	Years of experience	Skill origin
P1	Automotive	Engineering service	>1000	4	training-on-the-job
P2	Automotive	Technical leader	>1000	9	training-on-the-job
P3	Avionics	Systems engineer	>1000	10	university/education
P4	Automotive	Test Manager	21–100	24	training-on-the-job
P5	Avionics	Chief Software engineer	>1000	18	training-on-the-job
P6	Automotive	Senior Software Engineer	>1000	5	training-on-the-job
P7	Robotics	*No answer*	>1000	7	training-on-the-job
P8	Automotive	Embedded Software Developer	101–1000	4	university/education
P9	*No Domain*	Technical Lead of Functional Safety Software	>1000	6	other
P10	*No Domain*	Software Developer	>1000	10	training-on-the-job
P11	Automotive	Technical Leader	101–1000	6	self-taught
P12	Automotive	Project Engineer	21–100	8	training-on-the-job
P13	Automotive	Senior Software Engineer	>1000	4	training-on-the-job
P14	Automotive	CAE Analyst	>1000	2	self-taught
P15	Automotive	Tool support and design of modeling patterns	>1000	5	university/education
P16	Automotive	SW Developer	101–1000	8	training-on-the-job
P17	Automotive	Product Owner	21–100	3	training-on-the-job
P18	Electronics	MBD Evangelist [self-reported]	1–20	18	university/education

other port. Additionally, buses allow for signal hierarchies by encapsulating elementary signals and other bus lines.

On the other hand, once signals are encapsulated into a bus; the outer view loses information. The number and name of individual signals are no longer visible. Signals of a bus may be unrelated, or some may never be used but still carried along the bus. In addition, modelers have to organize buses through bus creators and selectors, increasing the number of required blocks.

3.3.3 Study Design

Research Objective

Our research aims to understand better Simulink bus usage to propose guidelines on how and when to use buses for Simulink practitioners. To this end, our study is driven by the following research questions:

RQ1 What are the advantages of bus usage?

RQ2 When is it appropriate to use buses?

RQ3 When should bus usage be avoided?

RQ4 What are best practices for bus sizes?

Research Design

For this study, we devised an inductive-deductive research approach in a triangulation fashion [214]. The inductive phase is used to develop hypotheses, which serve as guideline candidates. To develop our hypotheses, we conducted an exploratory survey with Simulink practitioners, analyzed open-source Simulink models, and derived them from prior literature. As our developed hypotheses could be invalid [58], we test them in a subsequent deductive phase. Here, we conducted a confirmatory survey with domain experts to cross-validate our hypotheses.

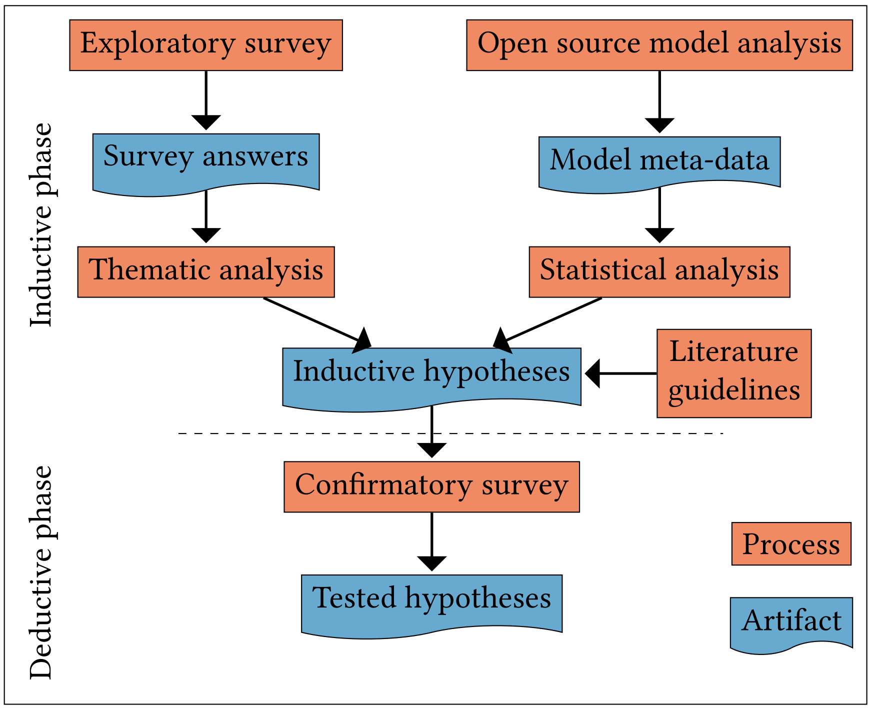


Figure 3.17. Two major phases of our research workflow: (1) the inductive phase where a survey analysis and model analysis result in the development of hypotheses, and (2) the deductive phase where these hypotheses are tested.

Inductive Phase

In the inductive phase, we followed an exploratory research approach [137] to identify bus usage advantages, disadvantages, and best practices. The phase is composed of three steps; (i) an exploratory survey conducted through a questionnaire composed of open-ended questions, (ii) an analysis of a curated corpus of open-source Simulink models gathered from public repositories, and (iii) a collection of guidelines from the literature (see Figure 3.17). The following sections describe how each of these steps was conducted.

Exploratory Survey We created an online survey with open-ended questions using Google Forms, designed to answer research questions RQ1, RQ2, and RQ3. Three sampling methods were used to reach out to participants. Voluntary response sampling was used through posts in eleven public forum groups from a professional social network[25] related to Simulink and model-based development (such as *"MATLAB & Simulink"* and *"MATLAB-Simulink and Model-based development"*). The survey was also distributed to the authors' contacts from industry and research (*i.e.*, convenience sampling), which were asked to share it further with relevant peers (*i.e.*, snowball sampling).

We first asked the participants five demographic and two qualification questions. The qualification questions were *Are you aware of the concept of buses in Simulink?* and *Have you ever used a bus in a Simulink model?* They were used to identify whether the participants

[25] https://www.linkedin.com (visited on 15/12/2025)

were fit to continue the survey. The demographic questions and their respective answer options were as follows:

1. What domain of application do you address in your Simulink models? (Energy, Electronics, Avionics, Robotics, Automotive, Health, Other [1])

2. What is your role in your organization? (open answer)

3. How many employees does your organization have? (1–20, 21–100, 101–1000, >1000)

4. How many years of experience do you have with Simulink? (open answer)

5. How did you acquire your Simulink skills? (self-taught, university/education, training-on-the-job, other)

Out of 20 responses, one participant never employed buses before the survey, and one mainly gave unclassifiable answers; these two were discarded from the further analysis (data in [342]). A summary of the remaining answers to the demographic questions can be seen in Table 3.10. The reader can use this table to trace the quotes that will be later presented in Section 3.3.4 back to the participants' backgrounds through the ID element. The participants' Simulink experience median is 6.5 years, and 8.4 years on average (see Figure 3.18). Moreover, Figure 3.19 summarizes the distribution of the domains in which the participants are working.

Finally, we asked the participants seven open-ended questions about buses and their application:

ES1: In which situations do you use buses in Simulink models?

ES2: In which situations do you avoid using buses?

ES3: In your opinion, what are the advantages of bus usage in Simulink models?

ES4: Describe a situation in which you experienced inadequate bus usage. How did this impact your work?

ES5: Does a model's size change how you work with buses? If so, please describe.

ES6: Do you differentiate between virtual and non-virtual buses? If yes, in which situations do you use virtual buses, and in which do you use non-virtual buses?

ES7: Under which conditions do you consider refactoring your model from several signals to buses or vice versa?

Thematic Analysis Having the survey results for the questions ES1 through ES7, we performed a thematic analysis [312] on the responses. First, we pre-processed the answers by removing those that participants did not develop far enough such that they could contribute to theory building, such as *"No idea"*. Then, we grouped the answers to each question according to their similarities. We created hypotheses for each of these groups according to the respective answers. For instance, we created Hypothesis H8, which states that *buses should be employed to improve efficiency when working on the model*, based on the responses to the question *RQ2 When is it appropriate to use buses?*. The following responses were placed together in the same thematic group, which inspired the creation of Hypothesis 8: *"to gain the ability to perform operations on the whole bus at once in the model"* (P9), *"[it] sometimes is easier to manage a bus structure to communicate software modules instead of updating the interface and adding inputs/outputs to the models"* (P6), *"when it becomes impractical to keep adding inputs outputs"* (P6), and *"consolidating signal inputs to scope"* (P14). Finally, we had a list of hypotheses traceable to the survey answers. In section Section 3.3.4, we provide the hypotheses and respective responses that sourced their development.

Open Source Model Analysis Some information types can be more readily and accurately retrieved from artifacts, especially for quantitative information, *e.g.*, the number of elements bundled in a bus. For answering RQ4, we first collected a set of 4,812 open-source Simulink models on GitHub stemming from 317 different projects [342]. To this end, we mined GitHub with Google BigQuery [378] for Simulink projects. While Kalliamvakou *et al.* warn of the perils of mining GitHub [149], Boll *et al.* [1] found open-source Simulink models to be diverse and suitable for empirical research, provided one uses an appropriate subset of models, *e.g.*, by removing ad-hoc and toy models. Thus, we constructed a subset of 433 from the original 4,812 models that employed at least one bus (bus creator block or bus output port) for further analysis.

Statistical Analysis We studied several metrics to quantify how buses are used, which allowed us to differentiate between typical and atypical bus usage. Our rationale for investigating this direction is that although open-source models may not represent *best practice*, typical usage still represents *common practice*, employed by Simulink users. We defined "atypical" bus usages as outliers below the 5th and above the 95th percentile of a metric's distribution and typical usage as everything in between (see Table 3.11). The metrics we computed per bus are:

- number of elementary signals: we recursively counted all elementary signals of all the sub-busses of a bus (minimum is 1)
- depth of bus: depth of the sub-bus tree (minimum is 1)
- number of sub-buses (minimum is 0)

We then formed hypotheses that atypical bus usage should be avoided or fixed. For instance, Hypothesis H18, which states that a bus should contain less than nine sub-buses, was created after we found that 95% of all busses have less than nine sub-buses. Thus, we consider having

Table 3.11. Summary of the distributions of our bus size metrics. We define "atypical" bus usage as below the 5th percentile or above the 95th percentile (numbers marked in peach). The minimum bus depth and the minimum number of sub-buses equal the fifth percentiles (the minimal possible values), so there is no lower cut-off. All metrics are long-tail distributions; thus, the averages are higher than the medians and include extreme outliers.

Per bus	**min**	**p_5**	**p_{25}**	**med**	**p_{75}**	**p_{95}**	**max**	**avg**
# of elementary signals	1	2	3	5	12	37	1392	15.49
bus depth	1	1	1	1	2	3	7	1.44
# of sub-buses	0	0	0	0	1	8	291	2.15

nine or more sub-buses as atypical. In section Section 3.3.4, we provide the hypotheses and respective statistical data that sourced their development.

Literature Guidelines From prior literature suggestions [172, 238, 261], we derived hypotheses on possible refactorings of problematic buses to further answer RQ4. For instance, hypothesis H22, which states that a bus should be made up of at least two elementary signals, was developed from the "superfluous bus signal" smell from Gerlitz *et al.* [172]. The value was derived from the 5th percentile of bus usage, which includes two signals. In section Section 3.3.4, we provide the hypotheses and respective literature references that sourced their development. These hypotheses fit our personal experience and reasoning, so we included them in the confirmatory survey.

Deductive Phase

Our goal in this phase was to test our previous findings through triangulation [214]. Inductive reasoning allows for the conclusion to be false [58]; thus, we tested our hypotheses with the help of a confirmatory survey.

Confirmatory Survey After eliciting hypotheses in the inductive phase, we tested them using a confirmatory survey (see Figure 3.17). The idea was to present our hypotheses to Simulink practitioners and ask them to state their level of agreement using a Likert type scale [21] of four points (*strongly disagree, disagree, agree, strongly agree*). Additionally, participants could decline to give a rating if they lacked the knowledge or were indecisive on a question (*don't know*). Thus, we employed a forced choice survey [62], *i.e.*, there was no *neutral* option. We distributed the survey using the same sampling methods and channels as we did with the Exploratory survey, plus among participants of the workshop *Buses in Simulink – A Blessing or a Curse?* organized by the Modeling Guidelines Interest Group [350]. It included the same qualification and demographic questions as in the Exploratory survey, described in Section 3.3.3. The survey received 36 responses, of which we excluded six: one from a participant answering twice, another answering every question only with *agree*, and four participants stated having no experience with Simulink buses. Finally, 30 responses remained for further analysis. Participants have a median of 8 years of experience and an average of 10.8 years (see Figure 3.18). The participants' domains are summarized in

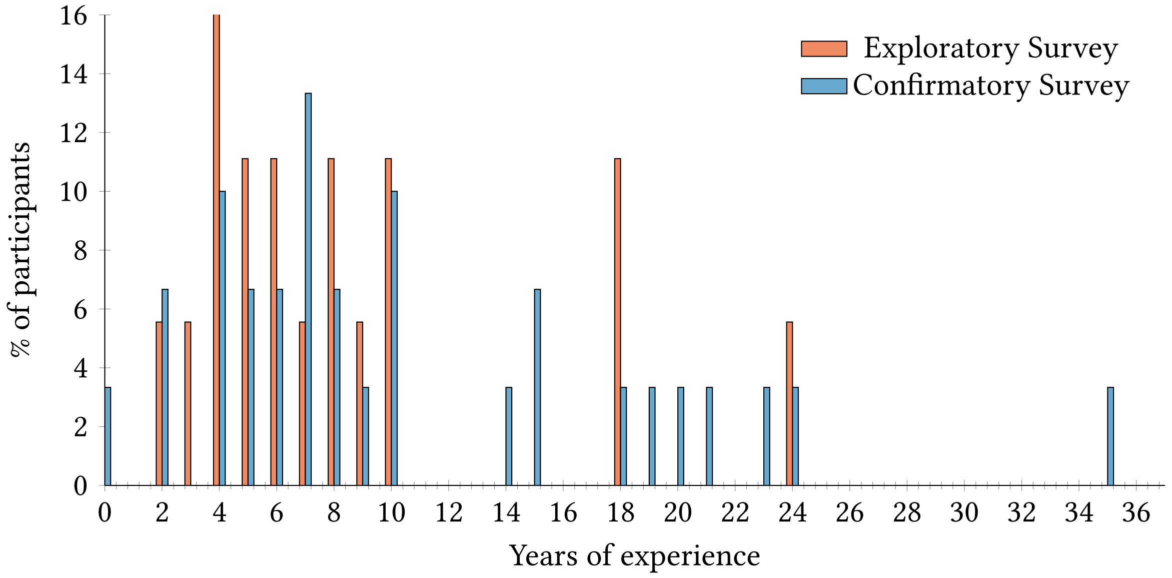


Figure 3.18. Distribution of the years of experience in working with Simulink of the participants of our surveys. The respondent with 35 years of experience clarified their answer "I worked on System Build, before Simulink existed."

Figure 3.19. Besides, the participants of the confirmatory survey show a similar diversity and distribution as the participants of the Exploratory survey (more details can be found in our data set [342]).

cycle list name = colors,

3.3.4 Results

Inductive phase results

Here, we present the 22 hypotheses developed during the Inductive phase of our study. We derived the hypotheses from multiple sources of evidence, namely the responses of the 18 participants of the Exploratory survey, the Open source model analysis, and the Literature guidelines. In the following, we describe these hypotheses and provide their source (*e.g.*, excerpts from the participants' answers).

RQ1 What are the advantages of bus usage? This section presents the hypotheses that address RQ1. They were elicited from the Exploratory survey; in particular, they were derived from the answers to question ES3.

Hypothesis H1. Buses reduce visual clutter in models. Generally, the participants value the very purpose of a bus to bundle together signals into a composite signal (*"combine signals" (P1)*, *"less signal lines" (P8)*) to diminish the number of visible elements (*i.e.*, lines and interfaces). In particular, models with fewer lines are considered to be cleaner, as indicated by notions such as *"cleaner models" (P10)*, *"cleaner interface" (P3)*, or *"more clear models" (P16)*, and to provide a better overview (*"better overview with less signal lines" (P17)*, *"much better overview in the models, less signal lines" (P15)*, *"better overview" (P9)*). In addition, some participants

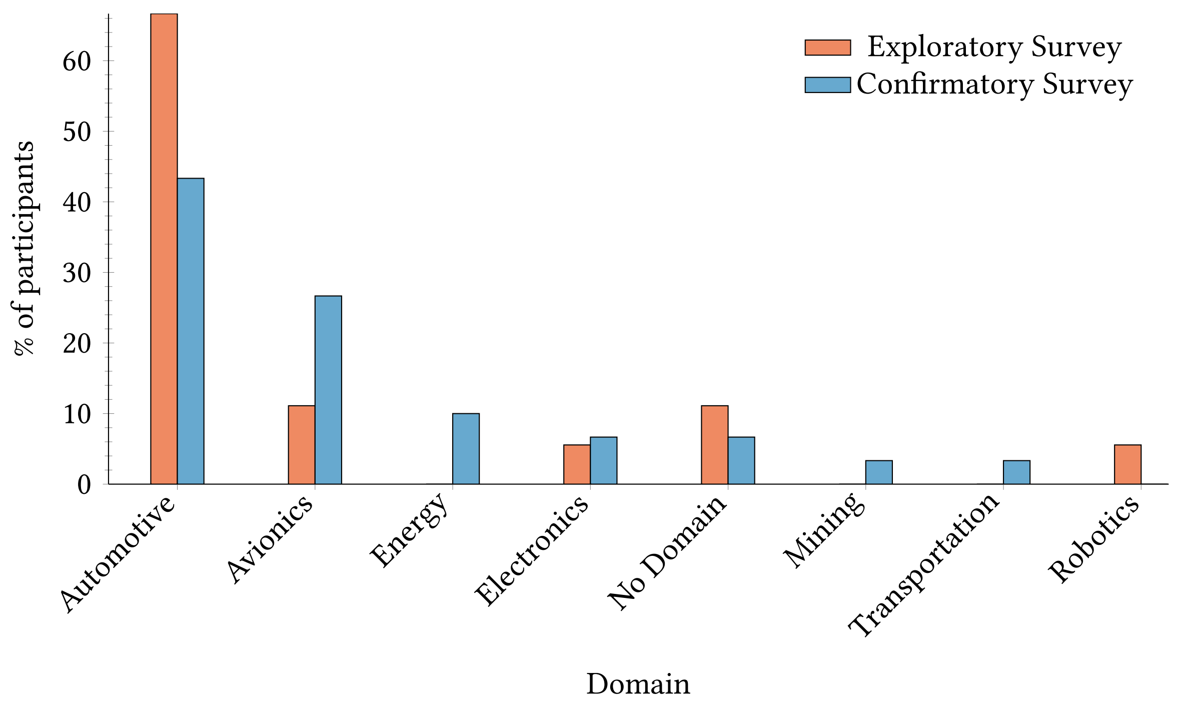


Figure 3.19. Distribution of the domains in which the participants of our surveys are working.

even mentioned that the reduction of the number of lines through using buses fosters *"simplification and flexibility" (P6)*, and that buses may help in *"reducing complicatedness in complex models" (P18)*.

Hypothesis H2. Buses are useful to provide engineers with information regarding signal relatedness through the model design. Buses can be used to convey information to engineers, who make assumptions about the data flow based on the signal visual flow [172]. Bundling signals in a bus suggests that these signals share a relation. Participants note that buses *"show constructively, which signals belong together (coming from the same source and mostly be used together)" (P4)* and that they enable a *"logical grouping of signals" (P9)*.

Hypothesis H3. Buses are useful to facilitate signal manipulation. A bus is a composite signal type that allows element access through names (i.e., like a struct or hash table), as opposed to index-based access. Thus, the elements' order is irrelevant. Such characteristic eases signal selection, *e.g.*, *"signal routing is made easier, and I don't have to care about the signals order like in MUX" (P12)*. When the signals are grouped in a bus, they can be shared between subsystems with less effort, as fewer diagrammatic operations are needed: *"less manual effort to connect signals one by one" (P13)*. Engineers can also *"perform operations on the whole bus at once" (P9)*, and can *"handle a set of signals" (P11)*.

Hypothesis H4. Buses are useful for configuring simulation and code generation. Non-virtual buses can be used for configuring model simulation and *"generating structs in code" (P18)* (*i.e.*, C-structs), which offers *"good cohesion with traditionally generated code through shared header files" (P5)*, as the *"structure in the code if non-virtual buses are used" (P17)* is preserved.

RQ2 When is it appropriate to use buses? This section presents hypotheses derived from the Exploratory survey addressing RQ2. More specifically, they stem from the answers to ES1, ES5, ES6, and ES7 questions.

Hypothesis H5. Buses should be employed when structuring diagrams. Many participants mentioned the architectural quality of buses. They noted that buses can be used *"as structure" (P1)*, *"to create structures" (P11)*, and to *"model architecture" (P9)*. It is preferable to have a *"structure of signals [for] too many scalar signals" (P17)*, or for *"multiple signals with different data types as structure" (P1)*. They can also facilitate encapsulation of subsystem communication and be viewed *"mainly as interfaces" (P7)*, or as an *"interface definition" (P17)* itself.

Hypothesis H6. Buses should be employed when grouping related signals (i.e., used by the same subsystem or function). The relatedness of candidate signals is a trigger for bus usage. Participants advise to *"couple related signals" (P3)*, *"to structure data that belong together" (P10)*, and employ buses *"when grouping similar signals" (P14)*, or *"when they have a strong dependency on each other" (P10)*. The modeled physical entity also influences bus usage. Some participants use buses when they *"reflect [a] real data network" (P3)*, and even more explicitly choose to *"combine CAN data and status signals" (P13)*. Two signals can be considered related if the same part of the system uses them, or they share the same function: *"if more than 2 data signals need to go through the same operation" (P2)*. In this way, buses *"transport signals, which belong together topically, between subsystems" (P8)*, or are used for *"exchanging large signals between sub-assemblies" (P14)*. More generally, it may be enough if signals take a similar flow through the model. Participants use buses *"to gather large groups of signals and lead them through the model" (P15)*, *"to bunch the signals moving between modules" (P13)*, and *"when [signals] are used in a combination at multiple locations" (P10)*. One participant explicitly mentioned that a signal's value might not be critical and recommends *"routing grouped constants and/or variables" (P18)*.

Hypothesis H7. Buses should be employed to improve model comprehensibility. Upon a high density of signal lines, the understandability of the diagram becomes impaired. A participant noted that the *"comprehensibility of the model" (P4)* could be improved, as *"too many signal lines could be overwhelming" (P4)*. In this way, buses can be used *"to simplify drawings by decreasing the number of crossing signals" (P10)* and are *"consolidating signals, to get a better signal flow overview" (P9)*. In other words, buses should be used as the *"model gets confusing due to too many signal lines" (P8)*. Another important aspect of buses is *"to simplify interfaces" (P6)*, by *"reducing interface size" (P18)*. This aspect can be viewed as fewer signals flowing into inports or from outports, and thus buses *"are also used to reduce the number of ports between the subsystems" (P16)*. Buses generally also simplify the layout of subsystems significantly. Both in terms of understanding and creating the layout. Getting a good routing in Simulink with many lines is effort intensive.

Hypothesis H8. Buses should be employed to improve efficiency when working on the model. Signals can be manipulated in bulk when grouped in a bus. Engineers thus want *"to gain the ability to perform operations on the whole bus at once in the model" (P9)*. Buses also save the time required to update interfaces between communicating software modules. Participants described this as *"[it] sometimes is easier to manage a bus structure to communicate software modules instead of updating the interface and adding inputs/outputs to* *the models" (P6)*, or *"when it becomes impractical to keep adding inputs/outputs" (P6)*. Finally, *"consolidating signal inputs to scope" (P14)* is also facilitated by buses.

Hypothesis H9. Buses should be employed when defining C-structs to be created in the auto-generated code. Buses marked as non-virtual trigger the code generator to transform the containing signals into C-structs. Several participants mentioned this feature, as they note buses are *"also useful to interface with C-structs" (P6)*, and *"code generation of structs" (P18)*, or even the very general remark: *"all models use buses to generate/access to expected C-structs in the code" (P16)*.

RQ3 When should bus usage be avoided? The hypotheses presented in this section address RQ3 and were derived from the Exploratory survey. More precisely, they were developed from the answers to questions ES2, ES4, ES5, ES6, and ES7.

Hypothesis H10. Buses should not be used when the number of candidate signals for a bus is low. Engineers may avoid using buses if the number of candidate signals one considers bundling in a bus is too low. This is usually the case for *"models with few signals" (P18)*, or *"models with simple interfaces" (P6)*. The participants considered different numbers of signals as too low, though: in *"small models [with] not so many signals (many being less than 8 or a bit more)" (P4)*, when the *"number of signals is $\leq$ 3" (P8)*, or even *"when it's only one signal" (P14)*. Too few signals in a bus can even lead to bus refactoring by bus dissolution from *"bus to signals: when only single signals are used in deeper model levels" (P10)*. Hypothesis H22 revisits this aspect and suggests a concrete number for "too low".

Hypothesis H11. Buses should not be used when signals have the same data type; a MUX should be employed instead. This hypothesis is similar to one of the very few guidelines concerning buses of MAB and MISRA [296, 379]. One should use MUXes for the same typed data and buses for mixed typed data, which one participant also stated as in the case of *"same data type signal" (P1)*.

Hypothesis H12. Buses should not be used when the candidate signals are unrelated. This hypothesis is the reverse of Hypothesis H6, and most participants thus answered with a reversed response. One participant noted that *"if the combination of signals into a bus is unpractical,* e.g., *signals come from/go to different subsystems, signals don't belong together" (P16)*.

Hypothesis H13. Non-virtual buses should not be used when they affect code generation negatively. To optimize the code generation, non-virtual buses should be avoided, *i.e.*, there is an *"autocode optimization need" (P12)*. Unlike non-virtual buses, virtual buses reduce memory requirements by accessing and storing data non-contiguously, *"to avoid memory conception" (P11)*.

Hypothesis H14. Buses should not be used when testing models; signals should be used instead. Doerr *et al.* mentions that, in testing, all signals of a bus will be created, even when they are not used elsewhere [238]. This superfluous signal creation adds to the cognitive load of the tester. Two participants also noted *"a to be unit tested (referenced) subsystem shall only get the inputs it needs, to minimize testing and analysis effort" (P9)*, and *"for testing models, signals are better" (P11)*.

Hypothesis H15. Buses should not be used when conveying signal usage information. When located outside buses, signals and their respective names become more visible and can convey information to the engineer. This way, one can be explicit and use a lower abstraction level, *e.g.*, elementary signals. It is also easier to discern which signals of a bus are used [238], or more concretely *"to show on which signals the sub-model really relies on" (P10).* Participants do not use buses *"if [they] want to model very clearly how submodules interact with each other (through which signals)" (P6)*, and *"don't often remove buses, but if it benefits understanding at the higher level, [they] can select signals from the bus and route as scalars into the subsystems" (P18).*

RQ4 What are best practices for bus sizes? The hypotheses in this section were developed either from analyzing publicly available Simulink models or from suggested refactorings of [261] and [238].

Hypothesis H16. If a bus is becoming too big, it should be split up into its sub-buses. This hypothesis states a refactoring proposal for big buses. If a big bus consists of sub-buses, it could be resolved into its sub-buses (a more radical step would be exposing even the sub-buses recursively as elementary signal lines). This refactoring could aid the understanding and handling, as smaller, more manageable buses can now be worked on, and more information is explicit, see Hypothesis H15.

Hypothesis H17. A bus should contain less than 38 elementary signals. In our analysis of public Simulink models, we found that 95% of all busses have less than 38 elementary signals (see Table 3.11). Thus, we consider having 38 or more signals as atypical. A possible refactoring is splitting the bus into sub-buses; see Hypothesis H16.

Hypothesis H18. A bus should contain less than nine sub-buses. We found that 95% of all busses have less than nine sub-buses (see Table 3.11). Thus, we consider having nine or more sub-buses as atypical. A possible refactoring is extracting some sub-buses (see Hypothesis H16).

Hypothesis H19. A bus should be nested less than five bus layers deep. We found that 95% of all busses are nested less than four layers deep (see Table 3.11). Thus, we consider having four or more layers of hierarchy as atypical. There was a typo in our questionnaire: our hypothesis should have stated "less than four bus layers deep"; it is thus less strong and may have led to a higher agreement rate. Possible refactorings are splitting the bus into sub-buses, see Hypothesis H16, or flattening the sub-bus hierarchy by resolving sub-buses into their elementary signals.

Hypothesis H20. When less than 50% of a bus' signals are used, it should be split up. One intuitive refactoring proposal for splitting up a bus is excluding unused signals. We suggest refactoring buses in which too many signals are unused, as they inflate the bus, making it harder to understand and handle. One participant in the Exploratory survey stated the following suggestion: *"when only single elements of the bus are necessary in the sub-models" (P10).*

Hypothesis H21. If a bus is too small, its elementary signals should be used instead. Buses that group up a tiny number of elements could add unnecessary abstraction (see Hypothesis H15).

Hypothesis H22. A bus should be made up of at least two elementary signals. This hypothesis brings a value for the "superfluous bus signal" smell from Gerlitz *et al.* [172]. The value was derived from the 5th percentile of bus usage, which includes two signals (see

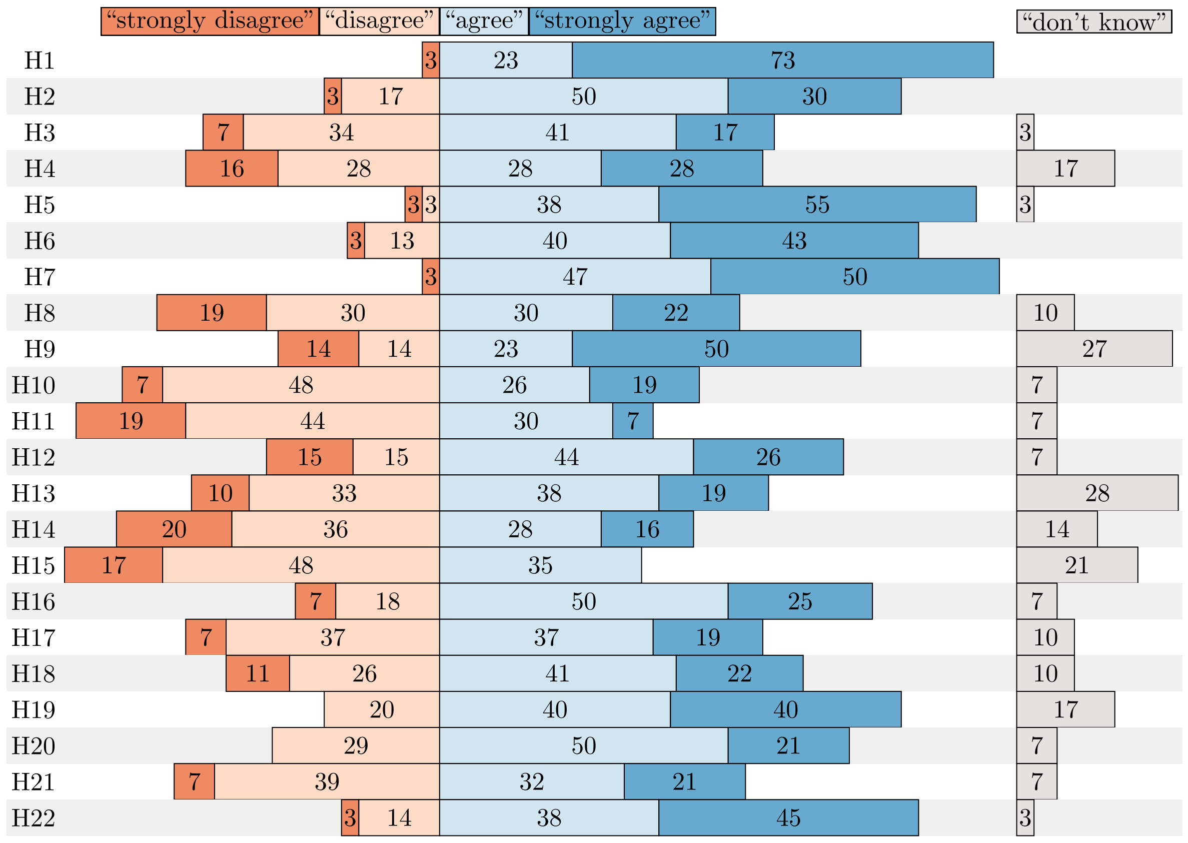


Figure 3.20. Confirmatory survey results. The colored bars in the middle show each hypothesis' agreement rate from the Confirmatory survey. The hypothesis IDs are denoted on the left side. The dark peach bars show the percentages of strong disagreement, light peach bars give the percentages of disagreement, light blue for agreement, and dark blue for strong agreement. Here, the "don't know" answers are excluded. They are shown on the right side with gray bars. Their numbers describe percentages of all responses with and without an agreement rating.

Table 3.11). Following this hypothesis, only buses using single elements must be refactored. As the minimum number of bus elements is one, this hypothesis is not particularly strong, but it gives a concrete number for the imprecisely defined "low" of Hypothesis H10.

Deductive phase results

The results of the Confirmatory survey are depicted in Figure 3.20. The numbers are given as percentages for the agreement level of the 30 respondents of the Confirmatory survey. We excluded the "don't know" answers participants gave, which we show separately on the right in gray color, as percentages of all responses. We did this because "don't know" responses cannot be rated on our forced-choice Likert type scale, see Section 3.3.3.

3.3.5 Discussion

In our discussion of study results, we first focus on the Confirmatory survey results, then discuss the results' applicability for industry and academia and finally consider potential threats to validity.

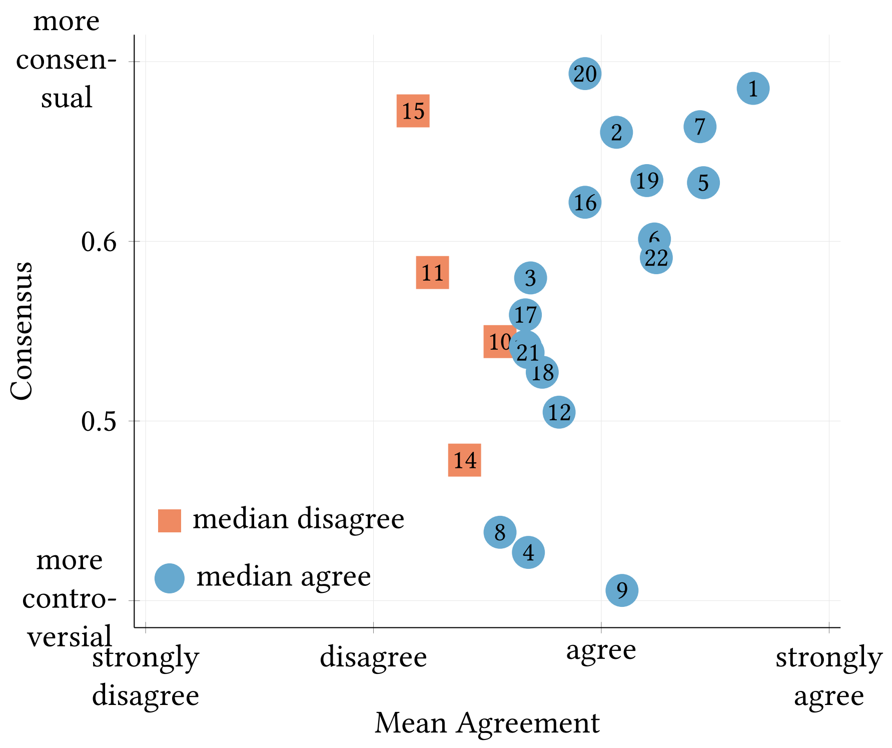


Figure 3.21. Mean agreement and consensus of hypotheses. Hypothesis H13 is located behind hypothesis H21.

Confirmatory survey results discussion

To discuss the Confirmatory survey results, we first introduce two dimensions, *consensus* and *mean agreement*, as criteria to analyze a hypothesis' acceptance. Consensus [71] measures the dispersion within an ordinal scale's answers. It is defined between 0 (complete dispersion between respondents) and 1 (complete agreement between respondents). The mean agreement is computed by converting the ordinal scale into an equidistant interval scale.[26] These values for each hypothesis are depicted in Figure 3.21, similarly to [310]. The x-axis shows the mean agreement value towards the hypotheses, the y-axis shows the consensus level of the participants. Color and shape disclose the median agreement for each hypothesis. In this chart, *don't know* answers were removed. For instance, hypotheses with a lower consensus are more controversial. They are located in the lower part of the chart (*e.g.*, H4, H8, H9). These may need some clarification to be universally accepted or apply only to specific contexts. On the other hand, hypotheses located on the upper part of the chart (*e.g.*, H1, H2, H7, H20) had higher consensus, probably because they are more generalizable and were well understood by the participants. For the mean agreement, hypotheses located next to the horizontal extremes have more *strong agreement* or *strong disagreement*. For instance, hypothesis H1 is located very much to the right side, with 73% of *strongly agree*. In the

[26] We are aware, that the responses are scaled ordinally and calculating a mean value for them is refrained from. Computing them for hypotheses with the same median values can still give an informative order of the level of agreement, though. We do not interpret mean values in our discussion.

following, we analyze the hypotheses, comparing them according to the responses received.

Hypotheses with high agreement rate All hypotheses from RQ1, RQ2, and RQ4 had a good agreement rate (*i.e.*, 73% on average). The hypotheses of RQ1 and RQ2 concern the advantages of buses and usage situations; thus, practitioners recognize buses' positive attributes and use cases. Perhaps the cut-off numbers in RQ4's hypotheses could have been stricter, as the agreement rate to bus size hypotheses was fairly high. In any case, bus sizes in the upper and lower fifth percentiles of open source model usage (see Table 3.11) were considered problematic by most respondents.

Hypotheses with high disagreement rate Four hypotheses received more disagreement than agreement: H10, H11, H14, and H15. They all address RQ3, which focuses on *situations when buses should be avoided*. Respondents could not agree on the hypotheses with a similar consensus to RQ2, which focuses on *when it is appropriate to use buses*. For instance, H10 was created based on the statements of six participants with a median of 7.5 years and an average of 10.5 years of experience. However, 55% of the participants of the Confirmatory survey disagreed with it. One possible reason for this outcome is the way the hypothesis was formulated (*i.e.*, *"the number of candidate signals for a bus is low."*) since most participants agreed that buses containing only two signals are too few (H22). Interestingly, H11 was the second most disagreed hypothesis, despite being elicited from the official guidelines. This fact may show the general need for founding or revising guidelines empirically based on actual practitioners' needs. Hypothesis H14 was derived from two survey participants, and we also find reference to it in the literature [238]. Perhaps, a rewritten hypothesis with more information, such as *thus, diminishing testing efforts* would have had a higher agreement rate. In H15, the least agreed hypothesis with only 35% and the only hypothesis without a single *strongly agree*, the wording of *"conveying signal usage information"* similarly could be open to misinterpretation. Disagreement with H15 was rather consensual, suggesting the opposite of H15 may find more acceptance: *buses should be used to convey signal usage information.* These hypotheses are good candidates for future work, where we try to understand the reasons for low adherence.

Hypotheses with high "don't know" rate Overall, the ratio of participants answering with *don't know* was low (9.2%), from which we conclude that our hypotheses are understandable and practitioners indeed have the knowledge and an opinion about them. Two hypotheses (H9 and H13) had more than 25% of *don't know* responses, and both are concerned with non-virtual buses, which we assume is a Simulink feature many engineers do not employ or have little knowledge of. This phenomenon could also explain why H9 achieved the lowest consensus in our study (see Figure 3.21). Despite many respondents being unable to provide a stance, the ones who did mostly agree with these hypotheses. Similarly, H4, where 56% of the respondents have agreed, shows that buses are used relatively little for code generation. Having notions of "simulation" and "code generation" in the same sentence could have confused the respondents due to their relation, influencing the high rate (17%) of *don't know* answers.

Applicability of our findings

This section discusses our findings' applicability to industry and academia.

Impact for Industry The bus element, when properly employed, lowers an engineer's cognitive demand by increasing internal model quality [101]. This may affect static properties, such as analyzability, modifiability, and testability. Poor internal quality, and smells [42] can contribute to increased technical debt [180]. In this matter, hypotheses H6 and H9 can serve as new guidelines on when to use buses. H12 and H13 guide when to avoid bus usage. H16 and H21 give concrete refactoring guidelines for buses with suboptimal signal amounts. Additionally, H17–H20 and H22 present concrete and empirically founded antipatterns. Detection of these cases is a good candidate to be implemented in static analysis tools and resolved through one of the refactorings proposed in H16 or H21.

For Model Engineering Solutions GmbH (MES), our industrial partner, the findings became the input for creating a static analysis tool prototype to give engineers hints on where buses can be used sensibly. This prototype represents the first step before incorporating these functionalities into their commercial tool. Overall, their perception is that buses are not used as much as they should; thus, the best practices for recommending when to use buses are the most interesting ones from their point of view. Our findings are also used in their Simulink teaching classes for system engineers.

Impact for Academia Hypotheses H1–H5, H7, and H8 empirically validate knowledge about the basic properties of bus usage. These describe conceptual grounding knowledge relevant to engineers new to Simulink. They describe benefits and define use contexts. This knowledge can also be incorporated into higher education course programs. Our findings contribute to the body of knowledge of design patterns in visual programming languages [162] with a strong focus on their graphical properties. Dataflow programming languages [119] (*e.g.*, ASCET-DEVELOPER [376]) bearing similar types of elements to the Simulink buses yields benefits from the results of our research. Some hypotheses and proposed guidelines may seem trivial, but they still help beginners by making tacit knowledge explicit and advanced practicioners by making the knowledge referable and consultable. We propose basing guidelines on empirical research, as H11 (based on a published guideline) was one of the hypotheses participants disagreed with the most.

Threats to Validity

During the inductive phase of our research, we used an online survey with open-ended questions, which could lead to misinterpretations of questions. To mitigate this threat, we carefully reviewed the survey before its execution, reviewed its responses after its execution, and discarded the ones that clearly showed some misunderstanding or did not provide helpful information for theory building. We posted our surveys in professional public forums (i.e., voluntary response sampling) and asked our peers to share them with relevant colleagues (i.e., snowballing sampling). Therefore, we had little control over who responded to our survey. This threat was mitigated through five demographic questions and two qualification questions, where the respondents were asked whether they had knowledge and experience using buses. We excluded respondents who never used buses in Simulink models from both surveys. With this exclusion criterion, we possibly also have excluded individuals who got a detailed explanation on why they should never employ buses. We presume, that most participants who have never used a bus, also don't know much about them, though. In any case, only one participant in the exploratory survey and three participants in the confirmatory survey may have been affected as they were excluded because of this criterion.

Webb [43] notes that a strength of triangulation is the cross-validation of hypotheses by different methods. Specifically, "When a hypothesis can survive the confrontation of a series of complementary testing methods, it contains a degree of validity unattainable by one test within the more constricted framework of a single method". However, once the different methods do not give converging results for a hypothesis, their degree of validity becomes questionable. This is why we only consider a hypothesis cross-validated by our confirmatory survey if its median level is in agreement, *i.e.*, blue circles in Figure 3.21. Still, this does not guarantee their validity – and symmetrically neither, that hypotheses H10, H11, H14, and H15 do not hold.

3.3.6 Conclusion

We proposed 22 hypotheses concerning Simulink bus elements, covering the advantages of bus usage, how and when to apply buses, and when they should be avoided. Eighteen of these hypotheses were agreed upon, while only four hypotheses received more disagreement. We used three methods to elicit our hypotheses: an open questions survey applied to experienced practitioners, an investigation of structuring decisions by studying real-life Simulink models, and a collection of guidelines from prior literature. In the second step, we applied a confirmatory survey to assess practitioners' agreement with our initial findings. Our findings help close the knowledge gap on these elements, which has not yet received enough attention from the scientific community. They serve as starting point for guidelines for Simulink practitioners and the development of smell detectors. In future work, we want to empirically compare the usage of buses to other abstraction and structuring methods of Simulink, namely the subsystem and MUX.

3.4 Case Study 3: Beyond code: Is there a difference between comments in visual and textual languages?

↪ Thesis Roadmap

Authors: Alexander Boll, Pooja Rani, Alexander Schultheiß, Timo Kehrer.
Published in: Journal of Systems and Software.

Abstract: Code comments are crucial for program comprehension and maintenance. To better understand the nature and content of comments, previous work proposed taxonomies of comment information for textual languages, notably classical programming languages. However, paradigms such as model-driven or model-based engineering often promote the use of visual languages, to which existing taxonomies are not directly applicable. Taking MATLAB/Simulink as a representative of a sophisticated and widely used modeling environment, we extend a multi-language comment taxonomy onto new (visual) comment types and two new languages: Simulink and MATLAB. Furthermore, we outline Simulink commenting

practices and compare them to textual languages. We analyze 259,267 comments from 9095 Simulink models and 17,792 MATLAB scripts. We identify the comment types, their usage frequency, classify comment information, and analyze their correlations with model metrics. We manually analyze 757 comments to extend the taxonomy. We also analyze commenting guidelines and developer adherence to them. Our extended taxonomy, SCoT (Simulink Comment Taxonomy), contains 25 categories. We find that Simulink comments, although often duplicated, are used at all model hierarchy levels. Of all comment types, Annotations are used most often; Notes scarcely. Our results indicate that Simulink developers, instead of extending comments, add new ones, and rarely follow commenting guidelines. Overall, we find Simulink comment information comparable to textual languages, which highlights commenting practice similarity across languages.

3.4.1 Introduction

Code comments (hereinafter comments) are crucial in helping developers understand, maintain, extend source code [29, 30, 257], and find locations of interest in the source code [88]. High-quality comments, therefore, have a high impact on lowering the development cost and improving the quality of software [56]. Given the importance of comments, researchers focused on many aspects of them, *e.g.*, automatically assessing comment quality [351], comment completion [341], comment generation [192, 240], to name only a few. Recently, researchers explored the contents of class comments and categorized the various information in them [226, 258, 320]. Building on this, Rani *et al.* [319] formulated a taxonomy of comment information, called Class Comment Type Model (CCTM), containing types such as summaries, warnings, recommendations, licensing information, etc. A complete taxonomy of comment types is needed for further automation of tools handling comments.

Prior categorization efforts, however, were done on textual class comments of object-oriented general-purpose languages (*i.e.*, Python, Java, and Smalltalk) only. On the contrary, little is known about commenting practices in language environments using visual paradigms, such as Simulink [276, 314]. In particular, the classification taxonomy from textual languages cannot be directly transferred. Apart from the different paradigms, Simulink has several ways to comment models, and the possibilities are more diverse than purely textual comments (see Section 3.4.2). Furthermore, Simulink models are often designed by non-software engineers [205]. Such domain experts may employ a unique commenting culture when compared with more “classical” software engineers.

Choosing Simulink as one *representative* of visual languages – Simulink is a mature software which is widely studied and employed in several key industries [97, 153, 159, 217] – our overall goal is twofold. We first aim at getting a better understanding of commenting practice in Simulink, before a comparison to textual languages shall help us to build a bridge for transferring existing knowledge from textual to visual programming languages. To that end, we study Simulink comments and develop a classification taxonomy for them, generalizing prior work to a visual language and its more diverse types of comments. Thereupon, we compare major characteristics of Simulink comments with those in the textual programming languages Python, Java, and Smalltalk.

In our study, we first extract a collection of Simulink comments from a large set of open-source Simulink projects [337]. Then, we study how Simulink projects are commented, which comment features are used, where comments are present in the model and for what purpose they are used. We manually classify a sample of our collection according to the existing CCTM taxonomy [319], and extend it to make it suitable for Simulink comments, yielding the Simulink Comment Taxonomy (SCoT). We also investigate, whether model size, age, or complexity correlate with a model's commenting effort. As Simulink projects often feature MATLAB code, we include the projects' MATLAB code in our investigations where appropriate. Then, we compare the commenting practices of Simulink and MATLAB with the practices of the previously studied languages [319], to gauge differences and similarities between them. Finally, we gather existing guidelines on MATLAB and Simulink and explore whether developers follow them.

The main findings of our study are as follows. The Simulink comment types are used in widely varying amounts, with Annotations being the most frequent comment type, while Notes are rarely used. Simulink comments are distributed evenly across all hierarchy depths, apart from the top levels, where developers clearly put in the most commenting effort. We found that size and complexity of a model correlate with the number of comments and amount of total comments of a model, but they do not correlate with the length of individual comments. This indicates that, as a model grows, developers do not add to existing comments, but create new comments instead. This underlines previous observations that Simulink comments, once created, hardly get revised [314] and also supports the claim that Simulink documentation becomes "rotten" [276]. Without adapting comments to an evolving model, developers risk that comments become out of sync with the model – which is a well-known concern from other programming languages [351]. We also found that Simulink and MATLAB comment information is highly similar in quality and quantity to previously studied comment information in Java, Python, and Smalltalk. The comments of all these languages cover mostly the same categories of our taxonomy, and these categories also show a similar distribution in all of them. This implies that, information-wise, the commenting cultures in Simulink and MATLAB are not much different from the textual languages Java, Python, and Smalltalk. We view this as an indicator that our extended taxonomy SCoT can be employed in the categorization of other programming languages. Similarly, we expect knowledge-transfer, regarding comments, between textual languages and visual languages and vice versa to be possible. Further, we believe that many of our conclusions generalize beyond Simulink to other languages and their tools. While analyzing the commenting guidelines of Simulink and MATLAB, we found only three, which developers rarely followed.

We summarize our contributions as follows:

- a qualitative and quantitative overview of Simulink commenting practices in a large and diverse set of open source projects and models;
- an empirically validated taxonomy, named SCoT (Simulink Comment Taxonomy), classifying the information of Simulink and MATLAB comments, also applicable for other languages;
- a comparison of Simulink and MATLAB comments to previously studied languages;

- a publicly available dataset of extracted comments and classified comments in the reproduction package,[27] as well as all scripts used in this work.

3.4.2 Background

Simulink

Simulink is a visual programming language developed by MathWorks.[28] Simulink offers a modeling environment for the simulation and analysis of graphical block-oriented models of multi-domain dynamical systems. It offers a high versatility through its many toolboxes for different scenarios and domains (*e.g.*, from theoretical simulation[29] to control of tangible systems,[30] in as different domains as solar power grids [97] to automotive [159]). Simulink is a widely used modeling language for industrial-scale cyber-physical systems [153, 217] and is widely studied by researchers [2].

A Simulink model is a data flow graph with vertices and edges. While the edges are represented as signal lines, the vertices are different kinds of blocks. Figure 3.22 shows two views of an example model with its blocks connected by signal lines. Each block of a Simulink model transforms its input signals into output signals, giving a data flow-oriented model. A signal's arrowhead next to a block signifies an input; the side without an arrowhead is an output of that block.

To manage the size and complexity of a large model, it can be divided hierarchically into subsystems. Each subsystem can contain further blocks, lines, and other subsystems, recursively. Simulink then shows the *view* of the model by only presenting blocks of the currently selected subsystem and hiding blocks nested in other subsystems. The model in Figure 3.22 has two views: the outer view with its subsystem highlighted in peach (Figure 3.22a) and the view from inside the subsystem (Figure 3.22b).

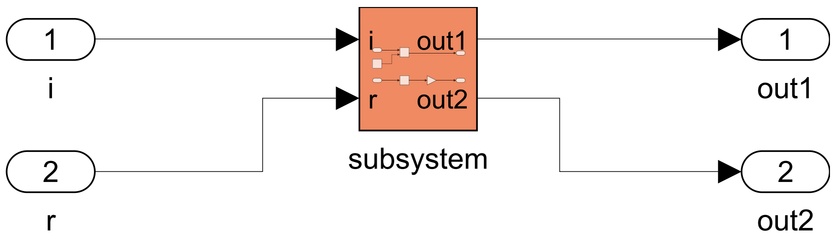
Simulink Comments

Some early research claimed that models do not need documentation because "models *are* documentation" and models are less ambiguous than textual documentation [54]. Today, however, the need for a model's documentation, has become clear [276].

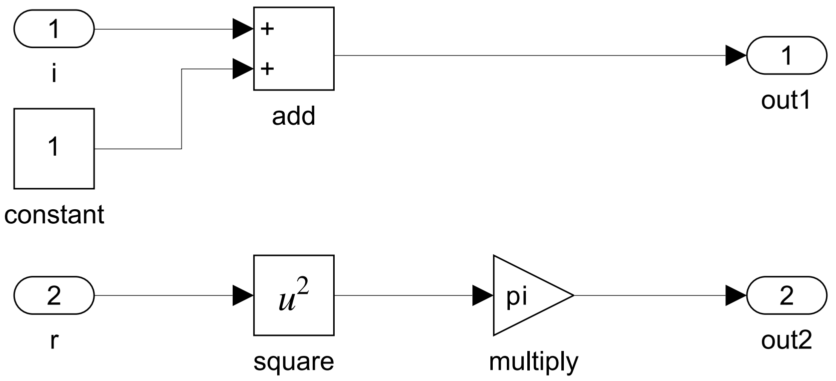
In this work, following the usual distinction between *internal* and *external* documentation [246, 277, 286], we focus on internal documentation directly integrated into the Simulink suite. Such documentation cannot get "lost" because it is in direct association with the model and will, by necessity, be as current as the model itself. Moreover, previous work on traditional programming languages has shown that developers embed various types of information in internal documentation [29, 30], which is often considered more trustworthy compared to all other sources of documentation (such as README files, user manuals, etc.) [154].

There are multiple ways of internally documenting Simulink models. At the time of writing, Simulink supports the following documentation types, which we will describe in detail below: *Model Description, Element Description, Annotation, DocBlock, and Note.* As

[27] https://doi.org/10.6084%2Fm9.figshare.24631350
[28] https://www.mathworks.com (visited on 15/12/2025)
[29] https://www.mathworks.com/help/mpc/ug/control-of-an-inverted-pendulum-on-a-cart.html (visited on 15/12/2025)
[30] https://www.mathworks.com/help/aeroblks/quadcopter-project.html (visited on 15/12/2025)



(a) The root subsystem view of the model. A subsystem is shown in peach, while its implementation content is hidden. The implementation content (see Figure 3.22b) is only hinted at on the subsystem symbol.

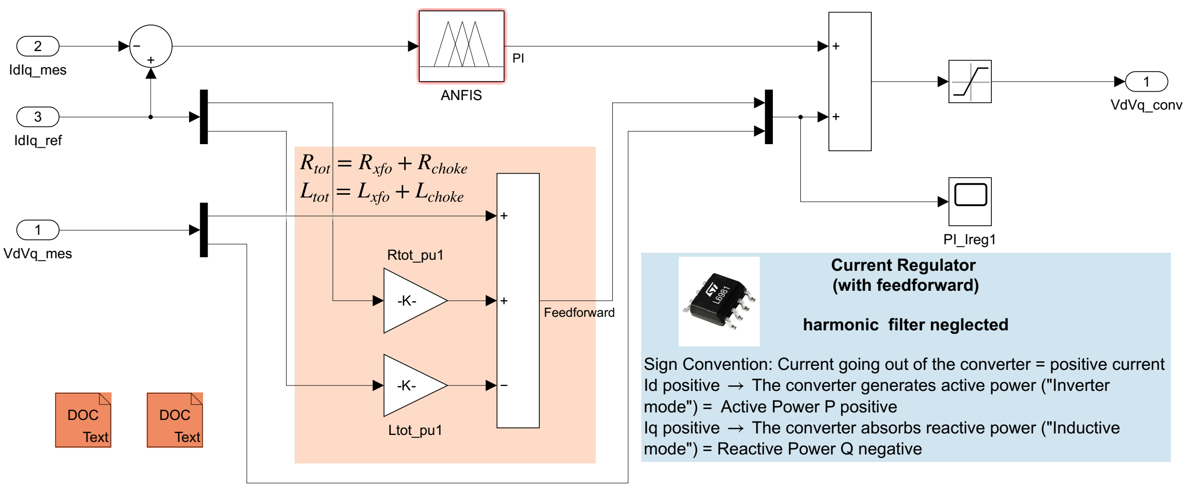


(b) The view from inside the subsystem reveals the detailed implementation through all its blocks and signals.

Figure 3.22. Two views of an exemplary model. The model computes the functions $out_1 = i + 1$ and $out_2 = \pi r^2$. The implementation of the functions is accessible and editable in the subsystem view in Figure 3.22b and hidden from the outside view in Figure 3.22a.

internal documentation in textual languages is usually referred to as *code comments*, we use the term "comment" for instances of internal Simulink documentation, even though they offer much more versatility than classical comments in textual programming languages. In the sequel, we still draw a comparison to textual comment types from both a *reference* and a *usability perspective*, so that the reader can get a better understanding. Before delving into the comparisons, it is important to note that these are not meant to be scientifically rigorous analyses. Instead, they are intended to offer some preliminary insights and intuition. A comment viewed from the *reference perspective* is the part or parts of a model the comment is about. Comments in textual and visual languages are thus comparable, if they reference comparable parts of a program or model, *e.g.*, a single code line and a single model element, or the whole code file and the whole model. The *usability perspective*, on the other hand, is about how developers are able to notice, access, and edit a comment.

Model Description: A model can be given a single, designated textual description, which is only accessible after four mouse clicks in a popup window from Simulink's menu, and is not displayed in the main graphical view of a model (see Figure 3.23b). Reference-wise, the closest analogy in classical programming languages are class comments or header comments, as there is only a single Model Description to describe the whole



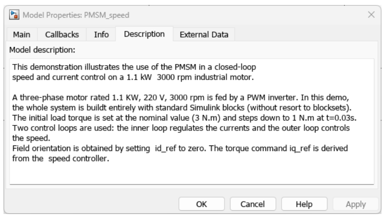
$R_{tot} = R_{xfo} + R_{choke}$
$L_{tot} = L_{xfo} + L_{choke}$

Current Regulator
(with feedforward)

harmonic filter neglected

Sign Convention: Current going out of the converter = positive current
Id positive → The converter generates active power ("Inverter mode") = Active Power P positive
Iq positive → The converter absorbs reactive power ("Inductive mode") = Reactive Power Q negative

(a) A model's subsystem, featuring several comment types: DocBlocks (on the left in peach) and Annotations (in champagne and light blue). The Annotation in champagne color highlights a specific area of the view, while the Annotation in light blue gives general information and shows a picture. Note the formatting opportunities, including LaTeX, in Annotations.


Model Properties: PMSM_speed

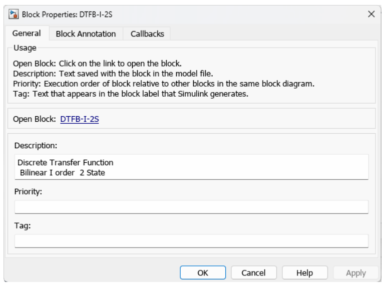
Main | Callbacks | Info | Description | External Data

Model description:

This demonstration illustrates the use of the PMSM in a closed-loop
speed and current control on a 1.1 kW 3000 rpm industrial motor.

A three-phase motor rated 1.1 KW, 220 V, 3000 rpm is fed by a PWM inverter. In this demo,
the whole system is buildt entirely with standard Simulink blocks (without resort to blocksets).
The initial load torque is set at the nominal value (3 N.m) and steps down to 1 N.m at t=0.03s.
Two control loops are used: the inner loop regulates the currents and the outer loop controls
the speed.
Field orientation is obtained by setting id_ref to zero. The torque command iq_ref is derived
from the speed controller.

(b) A Model Description's content is shown and edited in a popup window. There is only one Model Description per model.


Block Properties: DTFB-I-2S

General | Block Annotation | Callbacks

Usage

Open Block: Click on the link to open the block.
Description: Text saved with the block in the model file.
Priority: Execution order of block relative to other blocks in the same block diagram.
Tag: Text that appears in the block label that Simulink generates.

Open Block: DTFB-I-2S

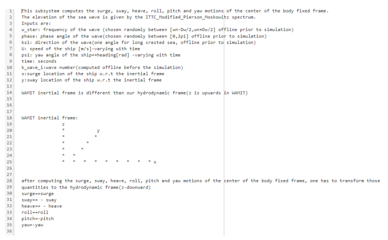
Description:

Discrete Transfer Function
Bilinear I order 2 State

Priority:

Tag:

(c) An element (block or signal) description's content is shown and edited in a popup window.


```
This subsystem computes the surge, sway, heave, roll, pitch and yaw motions of the center of the body fixed frame.
The elevation of the sea wave is given by the ITTC_Modified_Pierson_Moskowitz spectrum.
Inputs are:
w_star: frequency of the wave (chosen randomly between [wn-Dw/2,wn+Dw/2] offline prior to simulation)
phase: phase angle of the wave(chosen randomly between [0,2pi] offline prior to simulation)
ksi: direction of the wave(one angle for long crested sea, offline prior to simulation)
U: speed of the ship [m/s]-varying with time
psi: yaw angle of the ship==heading[rad] -varying with time
time: seconds
k_wave_i:wave number(computed offline before the simulation)
x:surge location of the ship w.r.t the inertial frame
y:sway location of the ship w.r.t the inertial frame

WAMIT inertial frame is different than our hydrodynamic frame(z is upwards in WAMIT)



WAMIT inertial frame:
               z
               *           y
               *          *
               *       *
               *     *
               *   *
               *   *   *   *   *   *   *   *   * x


after computing the surge, sway, heave, roll, pitch and yaw motions of the center of the body fixed frame, one has to transform those
quantities to the hydrodynamic frame(z-downward)
surge==surge
sway== - sway
heave== - heave
roll==roll
pitch=-pitch
yaw=-yaw

```

(d) DocBlock content is shown and edited in an external editor window. One may use, *e.g.*, Microsoft Word with its features.

Figure 3.23. Examples of Simulink comment types.

model. Usability-wise, the closest parallel in classical languages are README files or other external documentation, but Model Descriptions are an actual part of the Simulink model file.

Element Description: An element's description is associated to its model element (block, signal, bus). Users can describe the element, its usage, or context in more detail. An element's description text can only be accessed with two mouse clicks in a separate popup window (see Figure 3.23c). Reference-wise, we view Element Descriptions as most similar to inline comments, as they refer to single model elements, which are comparable to a short line of code. Usability-wise, there is no clear parallel we know of.

Annotation: An Annotation is a special area, placed in a model. These areas are mainly used to hold textual comments. They can also be colored and thus highlight a part of a model or even hold images. Annotations can also be linked to another model element, so the connection stays, even if the element is moved and the Annotation is not located nearby, anymore. Annotations are the only comment type of Simulink whose content is directly visible and editable in the model view. The champagne Annotation shown in Figure 3.23a highlights and explains a specific part of the model; the light blue Annotation gives the title of the view, further explanation, and shows various equations. There is also a small Annotation with a picture located on top of the light blue Annotation. Annotations can be used for model interaction, like holding a hyperlink to another subsystem or starting the model's simulation. Reference-wise, Annotations could be used like every type of code comment, due to their great versatility: a tiny Annotation next to a block like an inline comment, up to bigger Annotations describing a whole view or model, like a function or class comment. Usability-wise, they mimic all types of code comments, because of their immediacy, while additionally text formatting, pictures, and interactivity are possible.

DocBlock: A DocBlock is a special block in a Simulink model, which holds an embedded txt/html/rtf comment. As such, it can be used for longer and formatted comments. Two DocBlocks are part of the model in Figure 3.23a in peach color. Although the DocBlock, as a block, is part of the graphical model view, its text can only be accessed in a separate editor window (see Figure 3.23d), after a double click. Reference-wise, we view DocBlocks as most similar to function comments, because the DocBlock refers to a whole subsystem, which is comparable to a function. Usability-wise, DocBlocks work similarly to a clickable code comment hyperlink, which can be followed to some external documentation (this is sometimes used in, *e.g.*, JavaDocs), while the DocBlock and its content still is embedded in the Simulink model file itself.

Note: Simulink Notes are a mix of external and internal documentation. On the one hand, they are deeply integrated into the IDE. On the other hand, they are saved as external documentation files, only associated to a model file. A Note's textual content can be accessed with three mouse clicks in a separate editor window in the Simulink IDE, next to the model. Notes are more powerful than the other types, as they follow the model hierarchy. Depending on the current view of the model, a Note can show appropriate content only concerning this view. Thus, a single Note can be seen as a set of comments on classes or functions, reference-wise. Usability-wise, there is no clear parallel in classical languages. As our dataset lacks instances of Notes, we don't depict any in Figure 3.23.

MATLAB Comments

The MATLAB programming language uses textual representation for its source code. This means, the script files feature comments, similar to other textual programming languages. As Simulink models are often combined with MATLAB code in a project, we have the opportunity to study comments from bilingual projects in our work. MATLAB comments start from the %-symbol until the end of a line, or embrace comment text in between `%{` and `%}` brackets for multi-line comments. Listing 3.1 shows parts of a MATLAB source code file with several comments: the first multi-line comment from lines 1 to 4 gives a title and author information. The comments in lines 6 and 9 are short inline comments.

```
 1 %{
 2 Logger control for <project title>
 3 <Author Name>
 4 %}
 5
 6 % Initialize Log
 7 if enable_log
 8   set_param('Log','logging','on');
 9 else %enable_log == 0
10   set_param('Log','logging','off');
11 end
```

Listing 3.1. Examplary MATLAB source code showing a multi-line comment at the top and two shorter inline comments.

Simulink and MATLAB Comment Guidelines

We searched the official guidelines for High-Integrity Systems (G1),[31] and by the MathWorks Advisory Board (G2),[32] for instructions on how and when to comment in MATLAB or Simulink.

While G1 aims for "models that are complete, unambiguous, statically deterministic, robust, and verifiable", it does not provide advice on Simulink comments and gives only four guidelines regarding MATLAB comments. `himl_0001` requests to use a standardized header comment, `himl_0003` requests a comment density of 0.2 comment lines per line of code, `hisl_0038` asks for comment preservation in generated code, and `himl_0006`/`himl_0007` demands "meaningful" comments for if/else and switch statements. Note that the `else` statement of line 9 in Listing 3.1 is artificially commented by us.

One of the three aims of G2 is readability, which is further clarified as "improve readability of functional analysis, prevent connection mistakes, comments, etc." Still, we only found three guidelines related to Simulink documentation: `db_0140` display custom block parameters explicitly in the diagram, `db_0043` use consistent fonts and appearance settings across project, and `jc_0603` comment the model layer with a description. G2 also remarks that 'using Annotations [to group logically related parts as virtual objects] makes [the model] easier to understand'.

The Class Comment Type Model (CCTM)

Identifying the kinds of information embedded in code comments can support developers in various development and maintenance tasks, *e.g.*, an automatic comment classificator or updater would need a complete taxonomy of comment types. Therefore, researchers put a

[31] https://www.mathworks.com/help/pdf_doc/simulink/simulink_hi_guidelines.pdf (visited on 15/12/2025)

[32] https://www.mathworks.com/help/pdf_doc/simulink/simulink_mab_guidelines.pdf (visited on 15/12/2025)

lot of effort in classifying code comments, building code comment taxonomies. Based on taxonomies for textual programming languages, like Java, Python, and Smalltalk [226, 258, 320], Rani *et al.* [319] presented a taxonomy of class comments, called the Class Comment Type Model (CCTM). Rani *et al.* use the standard definition of classes in object-oriented languages, *i.e.*, classes represent blueprints for building instances [52]. Class comments are expected to hold various information [50, 226, 258, 319, 320], from high-level design to low-level implementation details [50]. The CCTM can be used to classify class comments into the following higher-level categories:

Purpose: A summary of the code's intent, further explanation of how the code works, or its rationale.

Notice: An explicit notice of exceptions, warnings, deprecation, or how to use the code.

Under Development: This encompasses development notes, notice of incomplete code parts or TODO-notes. It could also be commented code, coding guidelines or recommendations for extending the code.

Style & IDE: IDE or compiler directives or a comment that visually partitions code or comments into logical sections.

Metadata: Metadata could be licensing information, ownership information, or pointers to other resources.

Discarded: A higher-level category for comments that are not further analyzed: auto generated comments, unidentifiable (noise) comments, comments in a foreign language.

The six higher-level categories are divided into 20 lower-level categories: *e.g.*, the higher-level category *Purpose* consists of the lower-level categories *Summary*, *Expand*, *Rationale*. The complete breakdown of higher-level categories into categories can be seen in Table 3.16.

The CCTM is based on classifying comments from a diverse set of textual languages, which is why we assume some generalizability to comments from other languages, such as MATLAB and Simulink. Also, the CCTM offers a broad spectrum with 20 categories, which makes it currently the most fine-grained taxonomy [332]. Still, it is unknown whether the taxonomy can be directly transferred to Simulink or non-class comments in MATLAB. In this work, we use the CCTM as a first step to classify Simulink and MATLAB comments and complement it with missing categories to build our taxonomy SCoT, which is also applicable to non-class comments and visual languages like Simulink.

3.4.3 Methodology

Research Questions

The goals of this study are to explore the landscape of comments in Simulink projects, to understand how comments are used and what information they embody, and to establish a mapping of commenting practice in Simulink projects and textual programming languages. With this in mind, we design our research questions (RQs), and explain them in this section. Our focus is on Simulink models, as MATLAB is a textual language featuring comments that are similar to other textual programming languages. To put our findings for Simulink in context of those more well-understood languages, we also analyze MATLAB code from the bilingual projects of our dataset, similarly to Simulink, except for RQ 2.

RQ 1: How are Simulink projects documented?
There exist various types of comments in model-based development environments, such as Simulink (see Section 3.4.2). Not all types of comments are expected to be used in the same frequency. We give a breakdown of the usage frequency of Simulink's comment types. As Simulink models can consist of various subsystems (or layers of subsystems), the comments can also be present in various layers of these systems. However, whether certain layers tend to be more commented than others and with which comment types is unknown. We analyze this information at all levels of depth. During our work, we found many comments to be have identical comment texts (type I comment clones [311]), in some cases hundreds of times. We refer to such comments as *duplicates* and investigate possible duplication sources of heavily duplicated comments further. Finally, we investigate, whether developers follow the guidelines we collected in Section 3.4.2.

With RQ 1, we aim to answer, which comment types are typically present in models, learn their basic characteristics and where they are used.

RQ 2: Does the amount of documentation vary in different models?
Prior research searched for correlations between the amount of comments and other project characteristics in textual languages: *e.g.*, correlations exist between the number of comments and number of issues in the code [298], but no correlation between number of comments and number of project authors has been found [265]. However, to the best of our knowledge, it is currently unknown whether a model's age, size, and complexity and amount of comments show a correlation. With this knowledge, we can better gauge the importance of comments in big, mature, and complex models. Also, with such correlations established, comment smells [344] could be derived: developers should potentially revise the comments of strong outliers, *e.g.*, if a model grew very large but is still hardly commented.

RQ 3: How can the content of Simulink comments be classified?
As comments can cover many topics, *e.g.*, summary, usage tips, licensing information etc. we aim to understand, what they are employed for in Simulink and MATLAB. To this end, we classify Simulink and MATLAB comments, using the CCTM taxonomy by Rani *et al.* [319] from Section 3.4.2. We analyze the commenting practices in terms of what information is embedded inside different comments, such as *Summary*, *Warning*, *Copyright notice*, etc. Please note, the CCTM is a *Class* Comment Type Model. While our MATLAB samples feature a few class comments, most are in fact inline comments. Simulink, does not even feature classes, but offers various comment types (see Section 3.4.2). Thus, MATLAB and Simulink comment information may fall outside the current CCTM taxonomy.

Based on this step, we propose an extended taxonomy SCoT for MATLAB and Simulink that encompasses comments from textual and visual languages.

RQ 4: How does Simulink documentation compare to textual programming languages?
While the first three RQs focus on Simulink projects and exclusively on their languages Simulink and MATLAB, we also want to put these findings into context of previously studied languages. Simulink comments, with their various comment possibilities in a visual programming language, may differ significantly from textual programming languages. Depending on the results of our comparison, Simulink and MATLAB may have to be treated separately in documentation research or could be treated similarly to textual languages in some contexts.

Study Subjects and Data Collection

Data Set and Sample To collect Simulink model comments and MATLAB comments, we use the SLNET set by Shrestha *et al.* [337]. Their set contains 2,833 Simulink projects, consisting of 9,095 Simulink models (we could analyze 9,033 models successfully, *i.e.*, our analysis scripts ran error-free) and 17,792 MATLAB source code files. Shrestha *et al.* curated open source Simulink projects from GitHub and MATLAB Central.[33] The projects thus represent a highly diverse data set, comprising a range of tiny toy projects up to industry-like projects from various domains [1]. The SLNET set has been used in prior work for reproduction studies or learning about Simulink bus usage [4, 356]. We use the complete SLNET set to answer RQs 1 and 2, and have not excluded any comments, as we want to give a holistic overview of comments.

To answer RQ 3, we manually analyze a uniformly sampled subset of SLNET comments, as no automatic classifier exists for MATLAB or Simulink, yet. We thus choose the same sampling strategy as was used to create the CCTM taxonomy (see Section 3.4.2). We compute our sample size n, required to estimate population proportions of finite populations, according to the standard Equation (3.9) given by Triola *et al.* [61]:

$$n = \frac{\frac{z^2 p(1-p)}{e^2}}{1 + \frac{z^2 p(1-p)}{e^2 N}} \tag{3.9}$$

We choose our confidence level of 95% and thus the error $e = 0.05$, and $z = 1.96$. The value of p defaults to 0.5. We give a breakdown of sampled comments for each type of comment in the last column of Table 3.12. To get a better overview of the full breadth of comments, we deduplicate the SLNET comment set, before we sampled from it. After deduplication every comment has a unique comment text. This ensures that our results are not dominated by comments that are automatically generated, imported from libraries, or copy-pasted numerous times. We then use our manual analysis results of RQ 3 to answer RQ 4.

Extraction of Simulink Comments We analyze each model of the SLNET set element by element to check for the presence of comments (see Section 3.4.2). For each comment, we note relevant metadata, the main ones being the type of the comment (Element Description, DocBlock, etc.), the comment text and its length in chars, and the nesting depth in the subsystem hierarchy.

[33] https://www.mathworks.com/matlabcentral/ (visited on 15/12/2025)

In this first step, we found only 11 instances of Simulink Notes. As there are so few of them, we investigated them manually: five of them were automatically generated, the Simulink IDE was unable to load another five, and the last one was just a test Note. Because of this, we did not sample Simulink Notes for the manual analysis of RQ 3.

In the SLNET set, we found many duplicated comments (*i.e.*, comments with identical text). Based on Blasi *et al.* [311] and our observations, we suspect duplications coming from (i) a duplication process like copy-paste/cloning (individual comments, file duplications, or project forking), (ii) generic comments being located in multiple locations of a model (*e.g.*, copyright notice) or very short comments likely to appear more than once, due to the limited information they hold, (iii) library imports, (iv) generation by the IDE, and (v) synthetic generation. To not skew our results by heavily duplicated comments, we sample from a subset of deduplicated comments, only.

We further found that some comments stem from Mathworks' libraries or toolboxes. As they are part of the models – many toolboxes are open source projects in the SLNET set themselves – we do not exclude them from our sample. Due to the deduplication step described previously, such library comments are not overrepresented in our sample. Our sample set for manual analysis incorporates 374 Simulink comments. Table 3.12 gives an overview of the number of different comments, the cardinality of comment texts, and how many we sampled.

Extraction of MATLAB Comments In the 2,833 projects of SLNET, there are 17,792 MATLAB source code files. In 14,642 of them, we found at least one source code comment. For the manual analysis, we sample from the deduplicated subset, which results in 383 MATLAB comments. Table 3.12 gives an overview of the number of MATLAB comments, the cardinality of comments, and how many we sampled.

Computational analysis We extracted the Simulink comments and their metadata (see Section 3.4.3) directly from the models themselves with a MATLAB script. For this, we iterated over the whole model set, and within each model. We first collected a potential Model Description, all Annotations, and DocBlocks. Furthermore, we iterated over every model element and inspected it for a possible Element Description. We kept track of each comment and its metadata for further analysis steps.

We gathered the MATLAB comments using a Python script. We fused successive lines only containing comments to a single comment, even when the developers do not use the 'official' multi-line method of bracketing the comment between `%{` and `%}`. We did this, as the multi-line feature is not often used and developers tend to fall back to starting each line of their multi-line comment with a simple `%` symbol, even for very long comments.

All Simulink and MATLAB comments we found are gathered in `.json`-files, which we then analyzed further with Python scripts for RQs 1, 2 and 4.

Manual Classification Process To answer RQ 3, we first gathered the sampled comments into a shared Google sheet[34] for a collaborative classification process. Three researchers (a

[34] https://www.google.com/sheets (visited on 15/12/2025)

postdoctoral researcher and two Ph.D. candidates) participated in the classification process. We used the same three-step classification process as was employed by Rani *et al.* [319]: we split up the samples in a way that each comment is classified by one researcher in the first step. Next, another researcher reviewed the first classification and possibly proposed changes to the classification. The original researcher then accepted or rejected the proposed changes of the reviewer. If changes were rejected (if both evaluators disagree), a third researcher reviewed the comment and gave a final verdict on the classification. During classifying and reviewing, we kept track of missing classification categories, to expand or refine the CCTM taxonomy, by new categories, we observed. For example, Simulink contains some comments that have interactive features, for which we created a new *Interactive* category. For that purpose, all three researchers discussed their disagreements in the classification/reviewing process, as they are an indicator of the potential taxonomy refinement or extension. We also noted, how many comments needed a second or third review, to gauge our inter-rating conformity. This process yielded our taxonomy SCoT, in the same way as the taxonomy CCTM (see Section 3.4.2) was built.

In answering RQ 4, we use our findings of RQ 3 and compare the similarity of Simulink and MATLAB commenting practice with findings of studies that used the CCTM to classify Java, Python, and Smalltalk by Rani *et al.* [319].

3.4.4 Results

In this section, we describe the results of our study structured by research question; the discussion follows in the next section.

RQ 1: How are Simulink projects documented?

Table 3.12. The absolute number of each comment type found in the SLNET set for Simulink models and MATLAB source code files, is shown in the *comments* column. The deduplicated numbers are given in the middle column. The number of sampled comments for our manual analysis is given in the last column.

	Comment Type	*comments*	|*comments*|	*sampled*	
Simulink	Model Description	2,088	521	16	374
	Element Description	5,303	287	3	
	Annotation	91,027	11,348	348	
	DocBlock	308	129	7	
	Note	11	6	0	
MATLAB	Class Comment	472	354	3	383
	Other Comment	159,957	75,589	380	

General Measurement and Properties We counted the total number of each comment type in Simulink models and MATLAB source code files, and depict the results in the second column of Table 3.12. As can be seen, Annotations make up the overwhelming majority of Simulink comments, with over 90k instances in our 9,033 Simulink models. All other comment types combined only add up to about 7.7k instances.

Almost all MATLAB comments are non-class comments. In the 552 MATLAB classes of our source code files, we found 472 of the classes to have a class comment, though.

As can be seen when comparing the absolute (*comments*) and cardinality (|*comments*|) columns of Table 3.12, many comment texts are duplicated in our set (*e.g.*, around 88% of the Annotation texts are duplicates). From the class comments in our set, on the other hand, only 25% are duplicates, while over half of the non-class comments are.

Comment Duplication and Duplication Reasons To get a better understanding of comment duplicates (or clones) in Simulink and MATLAB, we present a scatter plot of duplicates in Figure 3.24. In the graph, the left-most comments are unique, while the right-most are heavily duplicated. The x, y-position of a marker represents that there are y different comments which are duplicated x times in our dataset. For example, more than 50k non-class comments from MATLAB are unique (dark blue marker at $x = 1, y = 54,287$), while the next marker at $x = 2, y = 11,598$ indicates that more than 10k comments of that type are duplicated exactly once; the last marker at $x = 1,524, y = 1$ represents one comment which was duplicated 1,523 times.

As can be seen in Figure 3.24, there are many duplicates (all comments with $x > 1$), with some comments duplicated dozens or in a few extreme cases more than a thousand times, such as Simulink Annotations or MATLAB's non-class comments. Such heavily duplicated comments are overall rare on the other hand, *i.e.*, the higher the duplication count of a comment, the lower the chance that there is another comment with a similarly high duplication count. This can also be seen at the sparsity of markers of most types for higher duplication counts. In fact, every comment type, except Element Descriptions, has more unique comments than those that have at least one duplicate. In other words: the first marker's y value of a type is higher than all the others combined.

To understand the duplication phenomenon better, we sampled the ten most duplicated comments of each category in Table 3.14 (represented by the right-most markers of each type in Figure 3.24). One can immediately see that some comments that occur most often are also highly similar, *e.g.*, the copyright notices in Model Descriptions: 1,773 of our 2,088 Model Descriptions are a MathWorks copyright notice.
For all comments of Table 3.14, we identified the duplication origins, *i.e.*, why the comment's text appears more than once. Based on our manual analysis, we hypothesized five types of duplication origins (based on [311] and our observations):

generic: a comment's text is very short or non-specific, making it likely that it appears more than once, *e.g.*, all Element Descriptions listed in Table 3.14,

copy-paste: the comment or the comment text was copy-pasted within the model or from model to model, *e.g.*, the most copied DocBlock of Table 3.14,

library: the comment is part of a library (only possible for Element Descriptions, DocBlocks, Annotations), *e.g.*, all Annotations of Table 3.14,

IDE generated: the comment or comment's text was generated via the IDE (*i.e.*, the IDE starts stubs for the user to fill in, or gives generic info), *e.g.*, "UNTITLED Summary of this class goes here \nDetailed explanation goes here,"

synthetically generated: we found a number of comments in MATLAB code that were synthetically generated. In fact, in all instances of synthetically generated comments we observed, the complete code files were synthesized, *e.g.*, "rad" and "Translation Method - Cartesian."

We often could not confidently categorize whether a comment was copy-pasted or just generic as we only observe the final identical texts and not the duplication process, and thus conservatively united the categories in Table 3.13. Only a few of the heavily duplicated comments are generated by the IDE or synthetically. Overall, one can see a divergence in the categories *generic/copy-paste*, *library*, and *synthetically generated* for the different types. The last column of Table 3.13 shows that taking only the top ten most duplicated comments, *e.g.*, Element Descriptions, represents already a high percentage of all comments of its type. This fact gives another perspective to interpret Figure 3.24.

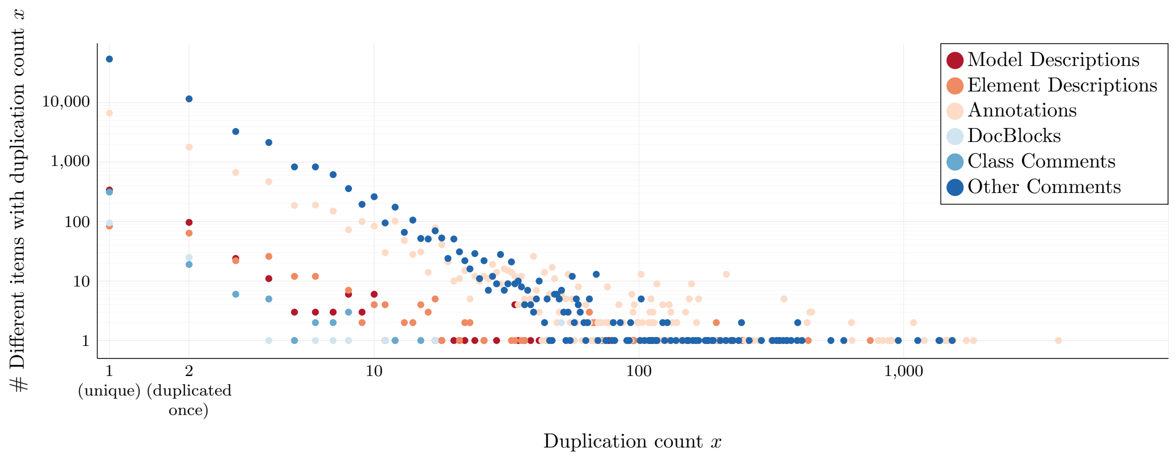


Figure 3.24. Scatter plot of duplication counts and the number of their occurrences. Note: both the x−axis and y-axis are logarithmic.

Comments at Different Levels of the Subsystem Hierarchy We give a breakdown of the commenting practices at different levels of depth of the subsystem hierarchy in Table 3.15. We define Model Descriptions to occur at the hypothetical depth 0 to include them in the table. One can see that most Element Descriptions are located at depth 4 and Annotations peak at level 3. DocBlocks are the only comment type with two local maxima at level 1 and level 7, respectively. In absolute terms, most comments occur at depth 3, while deduplicated, most comments lie at the root level (depth 1).

While the ratio of comments per subsystem is highest at the root level, the Model Descriptions at depth 0 lead to the highest ratio of comments per element. The ratio of comments per subsystems drops from depths 1 to 4, stabilizing afterward.

Table 3.13. Overview of the ten most duplicated comments' duplication reason per comment type. The last column shows the ratio of top ten duplicates and the total number of comments of that type.

Comment Type	Generic/ copy-paste	Library	IDE	synthetic	$\frac{\text{top10}}{\text{all}}$
Model descr.	10	0	0	0	0.42
Element descr.	4	6	0	0	0.59
Docblock	7	3	0	0	0.53
Annotation	0	10	0	0	0.18
Class comment	9	0	1	0	0.17
Other comment	4	0	0	6	0.05
total	34	19	1	6	0.11

Comparing the columns of *comments* and |*comments*| shows that the highest ratio of original comments can be found in the upper levels, with 75% of Model Descriptions and less than 50% of the comments at the root level being duplicates. At the other extreme are depths 10 or more, with > 98% duplicated comments, which is why we cut them from Table 3.15.

To not skew our analysis of comment lengths, we used only the deduplicated comments to compute the mean and median lengths in the last two columns. At all depth levels, the mean length (denoted by $\bar{x}_{len}$ in Table 3.15) of a comment is longer than its median (denoted by M_{len}), indicating a positive-skew (right-tailed distribution) of comment lengths. The mean length of Model Descriptions (depth 0) is much longer than any other comment. Similarly, the root level's mean comment length is about twice as long as on deeper levels of the subsystem hierarchy. Median lengths do not show a clear trend, with only the Model Description, again, being much longer than the rest.

As Annotations can both be containing text of various lengths, but can also be highlighting areas without text, we analyzed how Annotations are primarily used. We found that very few (0.2%) of the Annotations are highlighting an area only, *i.e.*, not holding a single comment text character. If used, such area-only Annotations are often highlighting a group of blocks (and not only empty model canvas). Overall, there are also few (8.1%) Annotations, containing one or multiple blocks, showing that most often Annotations are used as a purely textual companion, next to other model elements. Those Annotations that contained blocks usually hold a comment that is 10 to 100 characters long.

Comment Guidelines We investigated, whether developers followed the guidelines we gathered in Section 3.4.2. We skipped those guidelines that are not objectively measurable: guidelines concerning comment appearance and formatting, subjective guidelines about "meaningfulness" of comments; or guidelines regarding generated code, unobservable for us.

`himl_0001` (standard header comment) No source code file of our data set features the standard header of G1.

Table 3.14. The most duplicated comments in our data set listed by type. We marked shortened comments by [...], and new lines by \n.

Comment Type	number of occurrences	Comment Text
Model Description	247	Copyright 2017-2018 The MathWorks, Inc.
	117	Copyright 2014-2018 The MathWorks, Inc.
	95	Thomas Modules
	77	Copyright 2015-2018 The MathWorks, Inc.
	75	Copyright 2014-2017 The MathWorks, Inc.
	73	\nCopyright 2014-2016 The MathWorks, Inc.
	68	\nCopyright 2009-2018 The MathWorks, Inc. MathWorks, Inc.
	45	\nCopyright 2015-2017 The MathWorks, Inc.
	42	\nCopyright 2014 The MathWorks, Inc.
	39	\nCopyright 2013 The MathWorks, Inc.
Element Description	748	Initialise
	436	Output Signal
	342	Input Signal
	321	\nStore in Global RAM
	259	Add in CPU
	222	source block
	200	Trigger
	197	Fader Output
	196	Lower Limit
	196	Upper Limit
DocBlock	51	This subsystem computes the surge, sway, heave, roll, pitch and yaw motions of the center of the body [...]
	51	This subsystem computes the elevation of the sea wave, where the sea wave spectrum is given by [...]
	17	These are the Wave Excitation Forces computed by WAMIT-Demo version \nthe angle is between [...]
	11	jza - 21.08.07 The test model is created manually. \n\nTransformation rules for test data variants [...]
	8	Integral de sinal seno = sinal -coseno
	7	**Steps to Create a Quartus VHDL project****Simulink Steps**1. Setup all the paths
	6	Some text about the spec… \n
	5	Derivation of State Space model from original equations
	4	By testing SyD, you will be able to discover its advanced features and advantages
	2	Synchronous machine\r\n>>> Power conserving transformation
Annotation	3,840	The Measurement is not modified
	1,837	Pierre Giroux, Gilbert Sybille\nPower System Simulation Laboratory\nIREQ, Hydro-Quebc
	1,725	1) Only subsystems can be added as variant choices at this level\n2) Blocks cannot be connected at this [...]
	1,539	=
	1,464	Graphical user interface for the analysis of\nSimscape Power Systems \nPlace the Powergui block in the [...]
	1,434	*
	1,196	U(k)
	1,090	[d\n q]
	1,090	[al\n be]
	954	Integrator
Class Comment	15	Author: Colin Eles elesc@mcmaster.ca \n Organization: McMaster Centre for Software [...]
	12	Copyright 2014 The MathWorks, Inc.
	8	Author: Matthew Dawson matthew@mjdsystems.ca\n Organization: McMaster Centre for [...]
	8	Copyright (c) 2016, The MathWorks, Inc.
	8	%%% [...]
	7	Copyright 2014 - 2016 The MathWorks, Inc.
	7	CONNECTIVITYCONFIG PIL connectivity configuration class\n\n Copyright 2018 Arm Holdings
	6	%%% [...]
	6	UNTITLED Summary of this class goes here \nDetailed explanation goes here
	5	Copyright 2017 The MathWorks, Inc.
non-Class Comment	1,524	\n
	1,368	rad
	1,359	Translation Method - Cartesian\nRotation Method - Arbitrary Axis
	1,130	in
	954	m
	594	Las unidades de la resistencia son "Ohmios".
	531	User supplies all inputs
	431	kg*m^2
	398	Inertia Type - Custom\nVisual Properties - Simple
	398	%

`himl_0003` (comment density: 0.2 comment lines per line of code) We observed a higher mean of 0.271 comment lines per line of code, and a median of 0.25.

`jc_0603` (model description) We found only 2,088 of 9,033 models having a Model Description, while many of them are generic or copy-pasted duplicates, see Tables 3.12 and 3.13.

Table 3.15
Occurrences of Simulink comments (except Notes) at different subsystem depths.
[1] Mean and median lengths in chars of the row-wise deduplicated $|comments|$. [2] Each model is counted as one subsystem and element at depth 0 for Model Descriptions. [3] The only subsystem at depth 1 is the root subsystem.

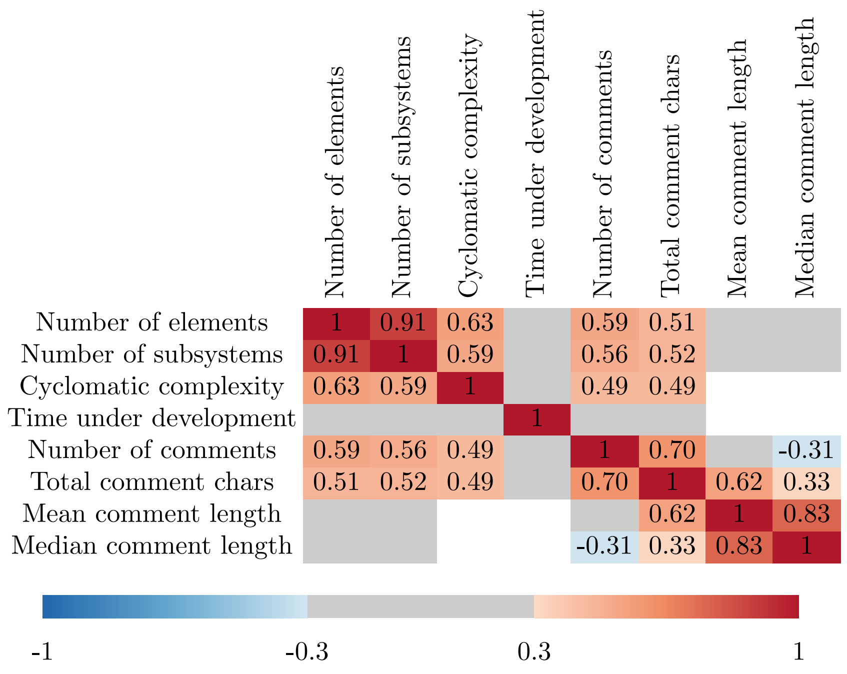
depth	Model Descriptions	Element Descriptions	Annotations	DocBlocks	*comments*	$\frac{comments}{subsystem}$	$\frac{comments}{elements}$	$|comments|$	$\bar{x}^1_{len}$	M^1_{len}
0	2,088	0	0	0	2,088	0.23^2	0.231^2	521	174.91	71
1	0	622	10,965	132	11,719	1.30^3	0.016	6,202	98.98	32.0
2	0	875	13,425	24	14,324	0.32	0.007	3,162	57.41	17.0
3	0	1,333	16,260	25	17,618	0.24	0.007	2,091	54.30	22
4	0	1,925	12,750	8	14,683	0.17	0.005	1,257	52.55	22
5	0	431	13,404	0	13,835	0.18	0.006	869	55.49	23
6	0	90	10,436	17	10,543	0.21	0.006	536	58.41	24.0
7	0	20	6,909	68	6,997	0.22	0.006	263	60.48	30
8	0	6	3,220	34	3,260	0.10	0.003	81	71.01	41
9	0	1	2,317	0	2,318	0.17	0.007	58	48.40	38.0
total	2,088	5,303	91,027	308	98,726	0.23	0.007	12,255	85.84	30

***RQ 1** How are Simulink projects documented?*
Annotations are the most used Simulink comment feature, while Notes are barely used. In MATLAB, there are few class comments. All types of comment show high numbers of duplicates, but each comment type (except Element Descriptions) has more unique comments than comments with at least one duplicate. Simulink models have the highest comment density at model and root level of the subsystem hierarchy for all comment types; the longest, least duplicated comments are also there. At lower depths, comments are often duplicated, but the density or comment length does not drop off. Few comment guidelines exist for Simulink or MATLAB; most not objectively measurable. MATLAB is commented more than guidelines demand, few models come with a model description, and the standard header is not featured in MATLAB code.

RQ 2: Does the amount of documentation vary in different models?
Here, we compute the correlation matrix of model size, cyclomatic complexity [197], and age as well as amount of model comments. We break down model size into overall number of model elements (blocks, signal lines) and number of subsystems, and use a model's age as a proxy for its time under development. We also break down 'the amount of comments' into number of comments, the total comment length in chars of a model, and the mean and median comment lengths of a model.

The correlation matrix of these metrics is given in Figure 3.25. As none of our metrics are normally distributed, we employ Spearman's rank correlation coefficient. We only consider higher correlations between two metrics, and ignore weak correlations $\rho < 0.3$ or too low significance levels of $p < 0.05$ (note: $p \neq \rho$).

Most correlations are significant: strong correlations are shown in color, weak correlations in gray. A few correlations are insignificant, shown in white. Only a single negative, albeit somewhat weak, correlation is present between the number of comments and the median comments' length of a model. There are only two comment metrics showing correlations to the model size, complexity, or age metrics: number of comments and total number of comment chars of a model. Lastly, time under development is uncorrelated to any other metric we measured.



	Number of elements	Number of subsystems	Cyclomatic complexity	Time under development	Number of comments	Total comment chars	Mean comment length	Median comment length
Number of elements	1	0.91	0.63		0.59	0.51		
Number of subsystems	0.91	1	0.59		0.56	0.52		
Cyclomatic complexity	0.63	0.59	1		0.49	0.49		
Time under development				1				
Number of comments	0.59	0.56	0.49		1	0.70		-0.31
Total comment chars	0.51	0.52	0.49		0.70	1	0.62	0.33
Mean comment length						0.62	1	0.83
Median comment length					-0.31	0.33	0.83	1

Figure 3.25. Heatmap of rank correlations of maturity metrics and comment amount metrics. Weak correlations with $|\rho| < 0.3$ are depicted in gray, and insignificant correlations with $p < 0.05$ are shown in white.

Spearman's correlation only measures correlations of ranks and not of actual values. This is why we also give an overview of the distributions of the metrics of maturity and comment elaborateness, which show a strong positive correlation in Figure 3.26. This shows, whether the values also grow somewhat similarly. For each of the metrics, we give the mean value of each quintile of their distribution. For example, if one sorts the models by the number of elements (the left-most five bars), the quintile of smallest models only has 11 elements in the mean, while the second quintile's models are bigger with 38 elements, etc. One can see that all metrics from our selection are strongly positive-skewed, as they grow from quintile to quintile even with our logarithmic y-axis. The last quintile features a "growth spurt" for all metrics. This "growth spurt" is especially drastic for the number of elements, subsystems, and comments. While none of the metrics is a complete outlier in terms of growth, one can see that the complexity does not grow as fast as the other metrics. Similarly, one can see that the total comment length does not keep up with the growth in the upper quintiles. Finally, this chart shows that most models only have a handful of comments, overall.

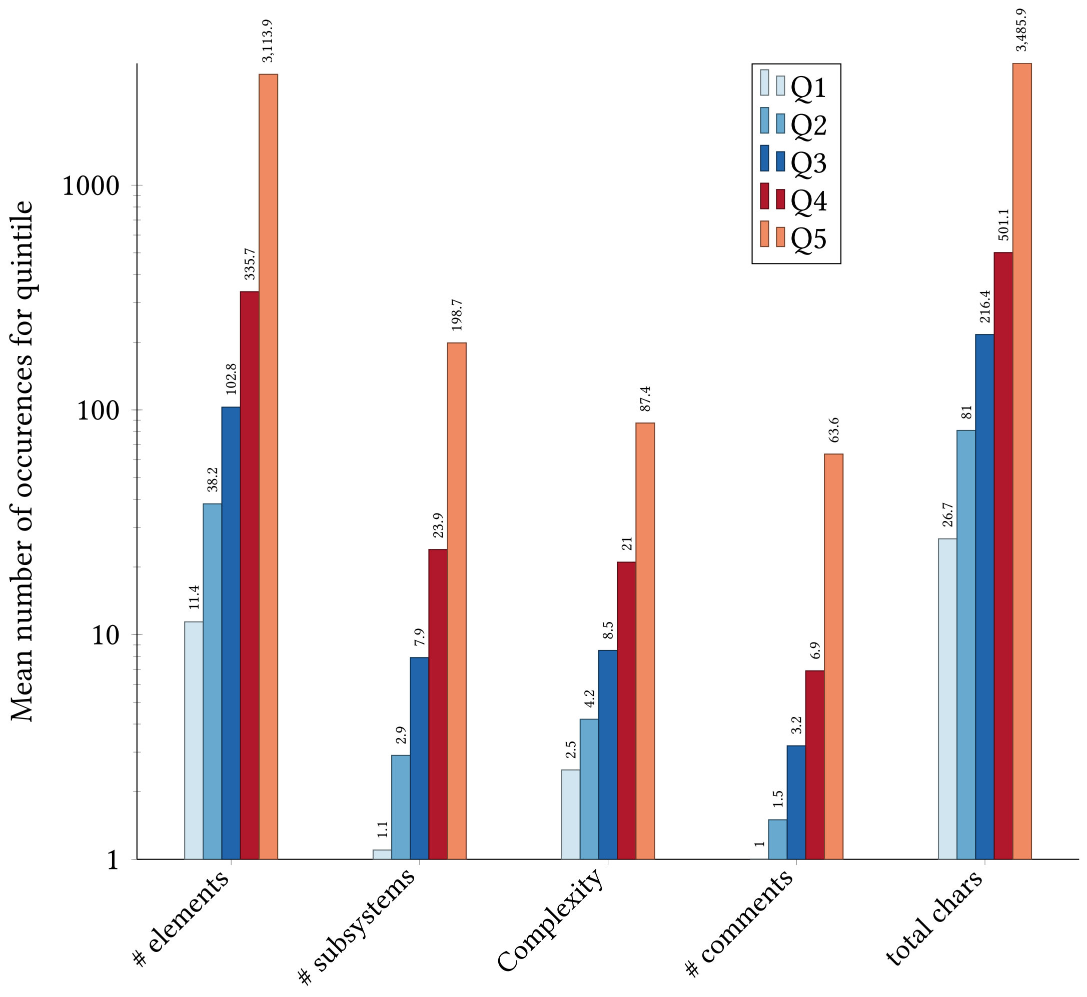


Figure 3.26. Mean quintile values of metrics that showed correlation.

RQ *2 Does the amount of documentation vary in different models?*
The number of total comments and total comment length of a model grows as the model grows in size (number of elements/subsystems) and complexity. Other correlations are either weak or insignificant. In particular, time under development does not correlate to any other metric we measured.

RQ 3: How can the content of Simulink comments be classified?

Deriving SCoT from CCTM While working on RQ 3, we started by adapting the CCTM's terms slightly to fit our context. This means that we changed terms like "source code" to "model" for Simulink, and adjusted terms of the CCTM only referring to "classes". We also decided on clear boundaries to differentiate between the categories *Summary* and *Expand*. In the CCTM, a *Summary* is a brief description of functionality and purpose, covering the question word 'what'. The *Expand* category is used to provide more details on the code to answer the question word 'how'. In practice, we found it hard to differentiate these categories and thus decided on a more objective criterion. To this end, we defined the category *Summary* to be a title of a module or summarizing at least 10 model elements or source code lines, while being at most 3 sentences long. We used the *Expand* category for the remaining candidates that were longer or described fewer elements/lines.

A few of our classified comments in Simulink and MATLAB did not fit in any prior CCTM category. To classify these comments accurately, we introduced five new categories (shown in italics in Table 3.16):

IDE Hint: (higher-level category notice) an instruction of how (not) to use the IDE to achieve certain results.

System Requirements: (notice) description/list of hardware or software requirements that make it possible to use the artifact and all its features.

Version History: (metadata) a description of older versions, version names, and dates of changes. This category is partly covered in the *Deprecation* category in the CCTM.

Interactive: (media) a comment which helps developers to interact with the program or IDE, such as interactive buttons in a comment that start or stop the simulation of a Simulink model.

Picture: (media) a picture, illustration, or figure for documentation purposes, such as a screenshot or example output.

Note that we created a new higher-level category *Media*, which is easily extendable for different kinds of media; other languages may use for documentation, *e.g.*, audio, video, etc.

While classifying, we came upon calls to action like "in case of bugs, please contact us at `adress@mail.host`". We expanded the *Ownership* category to cover such contact requests instead of creating a new category.

In the classification process, we decided to discard non-English text, as we could not ensure our complete understanding in categorizing such comments. We found text in Japanese, German, Dutch, and Spanish showing the diversity of the Simulink and MATLAB communities. As can be seen in Figure 3.27, the *Discarded* category was one of the smallest for both languages.

Simulink and MATLAB Comment Information The detailed results of our manual classification of Simulink and MATLAB comments are listed in Table 3.16. It can be seen that the lower-level categories *Summary* and *Expand* (from *Purpose*) are most often utilized, with the categories *Usage* and *Ownership* still being used for more than a tenth of comments. Overall, 22 categories are covered by our samples (19 by Simulink, 16 by MATLAB). We do not show the CCTM categories *Deprecation*, *Incomplete* and *Directive* in Table 3.16, as we found no instances of them in any of our samples.

From the 374 Simulink comments we analyzed, 59 covered more than one category. Many of such multi-topic comments were visually split into different parts by line breaks, where one part covered, *e.g.*, a *License Information*, followed by a *Summary*. Overall, we used 458 category classifications for our 374 Simulink comments (1.22 categories per comment). In MATLAB's 383 comments, on the other hand, 108 comments covered more than one category, totaling 630 categories (1.64 categories per comment).

An aggregation into higher-level categories of Table 3.16 is shown in Figure 3.27. While *Purpose* dominates across both Simulink and MATLAB, *Notice*, and *Style/IDE* still cover more than every seventh comment in each language.

Table 3.16. Detailed overview of the manual classification of our sample set: 374 Simulink and 383 MATLAB comments. The columns add up to more than 374 or 383, because a single comment can cover multiple categories. New categories of our taxonomy are printed in italics and unused categories of the CCTM are not shown.

Higher-level Category	Category	Simulink	MATLAB
Purpose	Summary	118	108
	Expand	131	235
	Rationale	11	17
Notice	Usage	93	56
	Exception	2	0
	IDE hint	2	0
	System requirements	0	5
Under Development	Development notes	11	17
	Todo	0	2
	Commented code	1	35
	Coding guidelines	1	0
	Extension	1	0
	Recommendation	1	2
Style & IDE	Formatter	6	30
Metadata	License	0	9
	Ownership	35	49
	Version history	6	19
	Pointer	16	25
Discarded	Auto generated	1	2
	Noise	15	19
Media	*Interactive*	6	0
	Picture	1	0

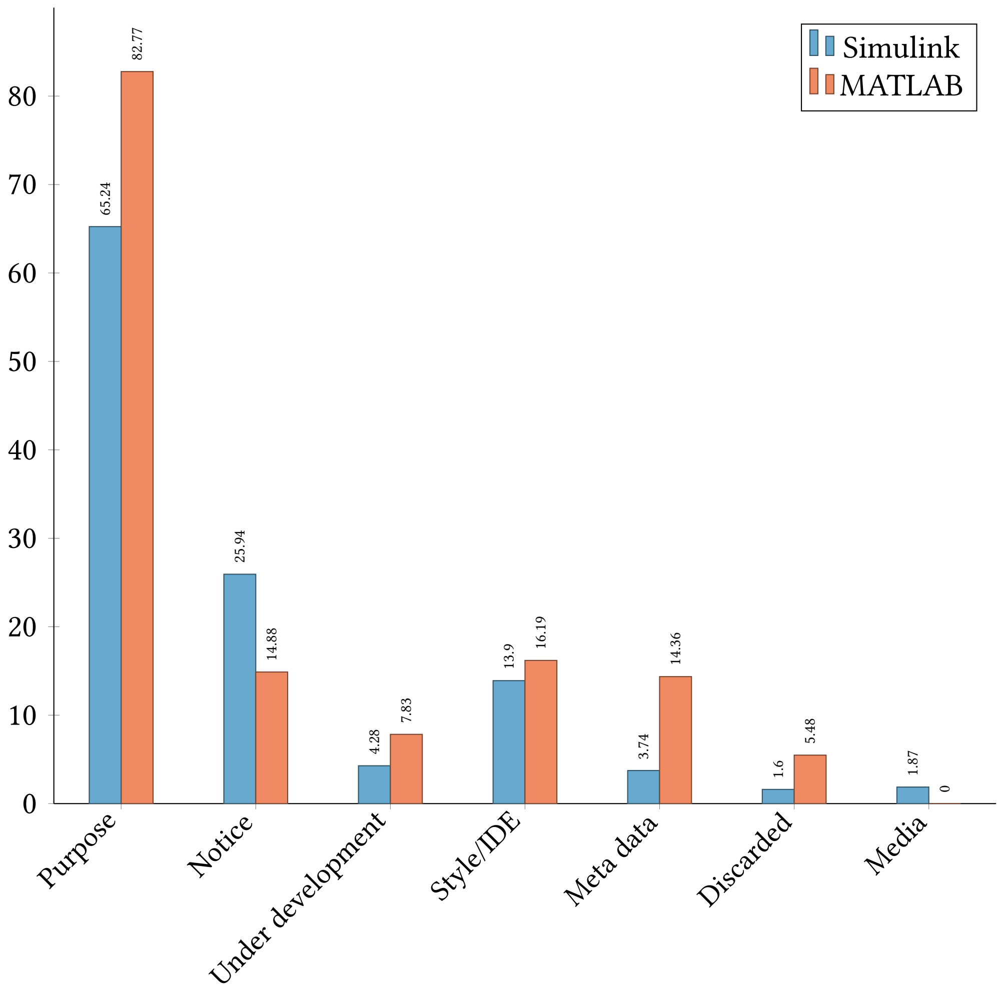


Figure 3.27. Higher-level category distributions of our sampled comments of Simulink and MATLAB. Note that the percentages sum up to more than 100%, because a single comment can cover multiple categories.

***RQ* 3** *Does the amount of documentation vary in different models?*
The CCTM taxonomy is mostly applicable to Simulink and MATLAB. We added the categories *IDE Hint, System Requirement, Version History, Interactive*, and *Picture*, while the categories *Incomplete Comment, Directive*, and *Deprecation* were not applicable. This yields our taxonomy SCoT. Simulink and MATLAB both cover nearly the full breadth of the CCTM taxonomy. Comments from the *Summary*, *Expand*, *Usage*, and *Ownership* categories dominate in both languages. Simulink comments are more narrowly focused per comment, as, on average, each comment cover only 1.2 categories, while a MATLAB comment covers 1.6 categories.

RQ 4: How does Simulink documentation compare to textual programming languages?

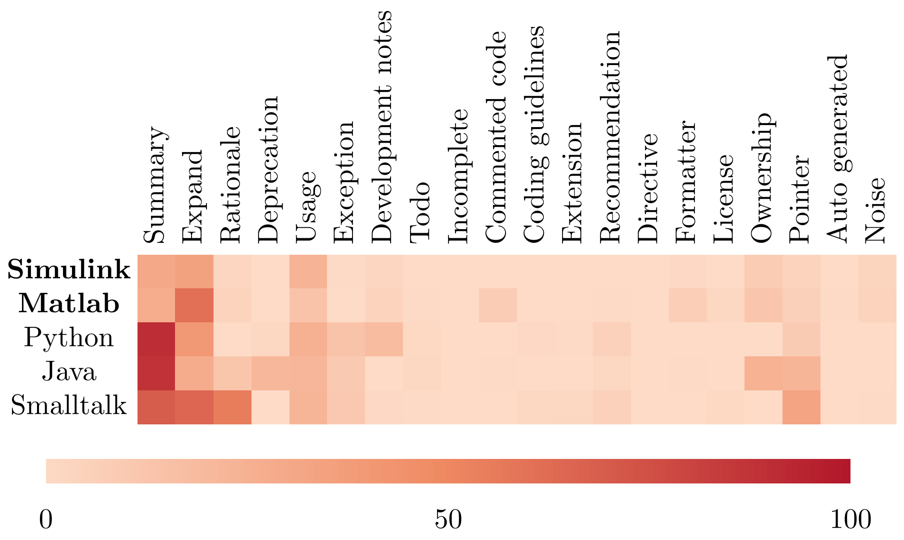


Figure 3.28. Heatmap comparing our CCTM categorization (highlighted in bold) and previously categorized languages. The categories of the CCTM are listed horizontally, while the color scheme depicts how many percent of comments fall into a category.

This question is partly answered by our answer to RQ 3: we employed the CCTM taxonomy, with only slightly adjusting descriptions of the categories and adding five seldom-used categories to the pre-existing category set. This shows that comments in textual and visual languages mostly cover the same categories.

A comparison over the distributions of classifications is shown in Figure 3.28. To make a comparison between the different languages possible, we use the category mapping found in Fig. 8 of Rani *et al.* [319]. The top rows of Figure 3.28 are very similar to a heatmap version of Table 3.16. The difference is that we use another category set in Figure 3.28: (1) we do not show our new categories as these were not part of the CCTM and could not have been found in Python, Java, or Smalltalk by definition, (2) we include categories that were used for Python, Java, or Smalltalk, which we did not find in our samples from MATLAB or Simulink. Overall, one can see a similar distribution between the languages, *e.g.*, the categories *Summary*, *Expand*, and *Usage* are heavily used in all languages. From the languages studied in this work, we found that they lack in *Exception* comments, compared to the other languages. MATLAB features more *Commented Code* and *Formatter*, than all other languages.

In RQ 3, we reported that, on average, a Simulink comment covers 1.21 categories, while MATLAB comments cover 1.64 categories. These results compare to the previously studied languages as follows: Python 2.23, Java 2.47, and Smalltalk 2.91 (derived from data of Rani *et al.* [319]).

RQ* 4 *Does the amount of documentation vary in different models?
Simulink and MATLAB comments cover mostly the same breadth of categories as Python, Java, and Smalltalk comments. In addition, each lower-level category was chosen similarly often for each language.

3.4.5 Discussion

In this section, we discuss our main findings, new insights, and possible implications, structured by research question.

RQ 1: How are Simulink projects documented?

General Measurement and Properties

We found that Annotations are the most common comment type in Simulink by far. This could be due to Annotations being the only type of comment showing the content directly in the model window. While adding a new Annotation, developers do not have to switch to another window and can use Annotations for several purposes (see Section 3.4.2) directly in the Simulink IDE. Readers of Annotations are also directly aware of the presence of Annotations and are able to read their content without opening a new window, as they would have to do with the other comment types. This impediment may explain the relative lack of instances of DocBlocks, Element Descriptions, and Notes. An additional reason for Notes is that they are the newest commenting feature in Simulink, only present since 2018, with the SLNet dataset [337] being gathered in 2020.

With 1,773 of 2,088 Model Descriptions featuring a MathWorks copyright notice in our data set of 9,033 models, we find the Model Description feature mostly unused, outside of MathWorks models.

In our view, some comment types in Simulink show usability shortcomings: comment types whose presence is not indicated to users immediately (Model Description, Element Description, Notes), or their content not directly accessible (DocBlocks) are hard to handle, or it is cumbersome to discover their existence. For example, users have to perform two clicks to see whether an element has a description, or not. We doubt that users would try to find out one by one which elements of a model contain an Element Description. A Model Description requires four mouse clicks to access, but there is only one Model Description per model, giving it a central place. DocBlocks are shown in the model, and users will thus see that some form of comment is present, but the content is opaque until accessed by a double click and waiting for, *e.g.*, Microsoft Word to open. Users may also need to install an `.rtf`-editor to access the content.

In view of all this, we suggest improving the accessibility of Element Descriptions by adding a small symbol on documented elements, or on a mouse-over to highlight the element or display the comment text. This ensures that developers become aware of an Element Description. DocBlocks similarly could display their (unformatted) comment on a mouse over, without opening an external editor window.

An alternative approach could be to refrain from using any other type of commenting feature apart from Annotations and Model Descriptions. This makes comments directly accessible in the case of Annotations, and gives a central documentation location to find and

automatically process vital model metadata in the case of Model Descriptions. Limiting the set of comment types could also help developers in their choice of which of the five comment types to use to document a particular aspect of their model.
While conducting this study, we asked Mathworks developers whether they view any of the commenting features as obsolete or to be preferred, in private communication. A Mathworks engineer disclosed to us that Mathworks views none of the comment types as obsolete, per se. The Mathworks engineer added that DocBlocks can viably be replaced by a Model Description, Annotation, or a Note, though.

We found class comments in 86% of MATLAB classes. This stands in contrast with previous findings [319]: 68% in Java, 23% in Python, and 38% in Smalltalk. As the class feature is seldom used in MATLAB (only 552 classes in 17,792 MATLAB source code files), it is an atypical phenomenon. This could explain the outlying percentage level. Note that we have checked that the MATLAB IDE does not create class comments automatically.

Comment Duplication and Duplication Reasons
Many comments in our study set are heavily duplicated, but different comment types are duplicated in different ways. For instance, we only found MATLAB comments to be synthetically or IDE generated. Some duplication actually is unavoidable, even with good commenting practice: Element Descriptions refer only to a single, often simple model element and, therefore, are expected to be more simple and similar to each other than more complex structures such as comments for subsystems or complete models. Following up on this, the high amount of generic/copy-pasted Model Descriptions seems to be haziness by developers: we found many simple copyright statements without any information concerning the specific model itself. We suggest that developers should follow guideline `jc_0603` (see Section 3.4.2), and be even more specific about the information of the model description: give at least a title and short purpose description of the model in addition to author and copyright information to each model.

Comments at Different Levels of the Subsystem Hierarchy
We found that the most elaborate, least duplicated comments occur at the root level of models. This is also the place with the highest comment density. Comment length does not change from the second layer downwards – only the frequency of comments at depths two and three is slightly higher than at lower depths. All this suggests that developers put more effort into documenting the top level(s). This might be because the root level and Model Description offer the possibility to document the complete model at once at a central place, which is easy to find. In contrast to this, lower-level comments may focus only on the direct context, *i.e.*, not the surrounding subsystems of higher or lower levels, and thus are shorter. Similarly to our findings, in Java, higher level comments (class, file, interface) have a higher density than method comments [339], while method comments are longer and show a higher comment density than the lower level inline comments [343].

Comment Guidelines
Our search for guidelines on documenting in Simulink and MATLAB returned only sparse results. In particular, novice developers would not be guided in most documenting decisions,

e.g., which elements to comment, what comment type to choose, or where to document. Of the three guidelines of which we tested developer adherence, only one was followed. We suspect that the official guidelines are not well known, or mostly ignored, by open-source developers. This suggests that developers employ their comments ad-hoc and comments differ from project to project.

We recommend giving clear advice on when to use which of the many Simulink comment options. We suggest investigating in more depth why developers currently mostly use Annotations and hardly use any of the other Simulink commenting features. We also suggest having one designated Annotation per subsystem for the subsystem *Purpose* (*Summary* and *Extend*) in a designated corner, *e.g.*, top left. This way, developers would know where to look for the most frequent information. One way to help developers and nudge them into employing such Annotations would be to automatically create this designated Annotation, partially pre-filled, at the very moment a new subsystem is created.

Our last guideline suggestion is to always attach an Annotation to a model element or a group of elements. If Annotations are tethered to another element, they cannot get lost or be forgotten about as easily if a model is refactored or otherwise modified (*i.e.*, documented elements are moved, copied, or deleted). Developers reading a diagram do not have the added burden of inferring which comment is referring to which element. Also, once an element and its comment have no connection (anymore), reattaching them presents challenges [198]. The title and purpose Annotations we proposed should then be tethered to a whole subsystem. To not overwhelm users with documentation text, we suggest to give annotations the new feature of minimization. This way, developers can elaborate design particularities or anything else at length, without cluttering the view canvas.

RQ 2: Does the amount of documentation vary in different models?

Answering RQ 2 gave us interesting insights into the (non-)correlation of various model metrics. For example, there is no correlation of model size or complexity to the length of comments. As models evolve, more model elements get added, than removed [10]. Only the number of comments and the total length of comments increase with models becoming bigger. Taking this together, it means that as a model grows, developers do not add to existing comments but add new ones instead. Based on our findings, it is unclear whether the existing comments get further updated, as their length remains the same. However, Jaskolka *et al.* found that Simulink comments are among the least changed elements of Simulink models in their industrial study [314]. This also mirrors findings in textual programming languages, where comments are not updated along with their corresponding code [284]. This fact sets comments up to be out of sync with its corresponding code or model.

Furthermore, we observed that a model's age is not correlated to any other metric of our study. This indicates that open-source projects either develop their models (not only comments) at very different speeds or do not consistently work on their models.

We saw a negative correlation between the number of comments and their median length (also a negative, albeit weak correlation to the mean length). This indicates that developers compensate for creating a higher number of comments by slightly shortening each. In our manual classification, we sometimes found short Annotations visually grouped tightly together, forming a connected documentation text if one unites the related Annotation texts of the group. Some developers used these individual Annotations to format a text, because each Annotation can be moved freely on the model canvas, so that it aligns to the developer's wishes.

We can see in Figure 3.26 that, in the upper quintiles, the number of comments and total comment length do not grow faster than the models themselves. This means that there is no relative increase in commenting effort in the biggest models. Before conducting this study, we had the hypothesis that bigger models are built by more professional teams, which would put more effort into commenting. This does not seem to be the case, at least in open-source models.

Other observations, like the correlations of size metrics and cyclomatic complexity, align with correlations of lines of code to cyclomatic complexity found in Java, C, and C++ [80].

RQ 3: How can the content of Simulink comments be classified?

When answering RQ 3, we found that the CCTM taxonomy covers a wide breadth of comments of Simulink and MATLAB, already. We also found it to be easily extendable. In our samples, only five seldom-used categories needed to be added to form SCoT. As our work did not focus on just class comments, but, in contrast to prior work, also considers models that are often designed by non-software engineers, we view this to be only minor additions. We thus expect the SCoT to be applicable in projects using other languages with only minor adjustments.

The most frequently used higher-level category from the SCoT for both Simulink and MATLAB is *Purpose*, showing that developers mostly care about documenting "what is the code about".

In our manual classification process, we found (and discarded) very few non-English comments. Even though, we did not classify them, we briefly analyzed them after an ad-hoc translation and found them to have similar information and format to English comments. We therefore believe that such non-English comments could be classified similarly to English comments.

Regarding our additional categories for extending the CCTM in RQ 3, we note that we only found few instances of the *Version history* category. We expect more sophisticated projects (as were studied prior with the CCTM) to usually handle the aspects of *Version history* either in release notes or directly in the VCS' commit history. We expect the new category *IDE Hint* to be used only for languages that use a common IDE. Both MATLAB code and Simulink models are commonly used in the MATLAB ecosystem, as a working and licensed MATLAB installation is necessary for their execution, anyway.[35] Lastly, the new higher-level category *Media* holds the Simulink-specific sub-categories *Interactive* and *Picture*. Classical programming languages are limited to text-based comments. However, previous work [277] has found media, such as images, in README files, which shows developer interest in expressing their documentation in different forms. In general, we expect other languages to enable commenting via media like audio or video in the future. Our new higher-level category *Media* can be extended with such modalities, easily.

[35] There is a plugin for MATLAB in Visual Studio Code, but only very basic features of code editing are available without a working MATLAB installation.

We imagine that comments of the *Interactive* category can be extremely useful in program understanding – both of abstract purpose and inner design. Various modes or parameters of the model can be preset, and their execution can be discovered immersively. Such interactive documentation thus offers the possibility to “show, not tell”.

While answering RQ 3, we manually classified each item. Prior work [198] already derived heuristics to identify some categories like Summary, Ownership, and Expand, for the diagram language Ptolemy,[36] with some success. We expect that similar heuristics could be divised for most of our categories. Similarly, we expect LLMs to be applicable for automatic classification, see also Section 3.4.7.

RQ 4: How does Simulink documentation compare to textual programming languages?

The findings from RQs 3 and 4 demonstrated substantial similarities in both quantitative and qualitative terms between Simulink and MATLAB commenting as well as Python, Java, and Smalltalk. This shows that Simulink, although a visual language with a diverse comment feature set, is, in fact, documented similarly to textual languages.

While comparing Simulink and MATLAB to Python, Java, and Smalltalk, recall that the prior studies focused on class comments from high-profile projects. This showed most prominently in that class comments covered more of the CCTM categories per comment than in our sample. This seems intuitive, as class comments are longer and more exhaustive than other code comments. In fact, we expect comparing class comments in Python, Java, and Smalltalk with MATLAB class comments and Simulink root subsystems’ DocBlocks, main Annotations, and Model Descriptions to yield similar results.

Quantitatively, the different languages showed a very similar distribution see Figure 3.28 – even though different research teams studied different languages, different comment types, and different project types. For us, this is an indication that commenting cultures are similar even while crossing so many boundaries. We thus expect that there is significant potential for knowledge transfer between findings from comments in textual languages to visual languages, and vice versa.

3.4.6 Threats to Validity

Internal Validity

Although our manual classification process for RQ 3 is subjective, we mitigate this threat by conducting a triple-review process with a majority vote and group discussions for unclear comments, similar to prior work [319]. By employing this technique, we strive for a more objective classification. A summary of our classification process is shown in Table 3.17. Around 20% (150/757) of the reviews objected that a comment’s category was missing, too much, or wrongly classified. In the second step, the original evaluators judged the reviews themselves and accepted about 75% of them. This left only 38 comments, where a third reviewer made a final decision after weighing both the evaluation and review. While the evaluation and review phase was evenly distributed by design, the steps afterward depended on the decisions of these two phases. For example, E3 had the highest agreement rate for

[36] https://ptolemy.berkeley.edu/ptolemyII/index.htm (visited on 15/12/2025)

reviewing MATLAB comments, *i.e.*, they issued only few objecting reviews to the original evaluation.

Some Simulink comments or MATLAB comments are part of a larger context of related comments. These are usually graphically close, or in a code line nearby. Our scripts to collect and sample comments could not link such "related" comments, and they were thus gathered in isolation. However, in our manual classification, we inspected each comment, and could thus see, whether a nearby comment was part of the context of our comment to classify.

By answering RQ 2, we found a model's time under development not correlating to any other metric, we computed. We hypothesize that Simulink may compute this time faultily in some cases. On inspection of the times, we only found 56 times from our 9,033 models to be obviously erroneous, though. These either had a negative time under development or one of less than ten seconds – we excluded them prior to our analysis in Figure 3.25. All correlations of time under development that are too weak (shown in Figure 3.25 in gray color), are positive. This indicates that the metric can be assumed to be correct, overall.

Table 3.17. Overview of the classification process. While 757 comments underwent an evaluation and review, only 150 of the reviews elicited objections, and of those only 38 were not accepted by the original evaluator and thus needed a final decision.

	Evaluator	evaluated comments	objecting reviews	final decisions
Simulink	E1	124	25	5
	E2	124	28	6
	E3	126	21	12
Matlab	E1	127	33	5
	E2	128	35	1
	E3	128	8	9
total		757	150	38

External Validity

Our analysis set consists of open-source projects from GitHub and Mathworks Central. Comments in industry-projects may differ significantly. Via industrial acquaintances, we know that some companies have internal guidelines but do not know whether these cover comments and how they would employ comments in Simulink. Still, our data set is highly diverse, comprising everything from toy projects to industry-like projects [1], and thus gives valuable insights into how Simulink comments are used in practice.

3.4.7 Related Work

Comment Analysis

Code comments are an active research topic which has evolved over decades. Already in 1976, Boehm *et al.* [25] started to develop metrics predicting software quality from quantitatively measuring source code commentary. In particular, they doubted that comment length alone is an indicator of good software. They also already gave advice of not over-explaining some code at the expense of leaving other code uncommented. Lastly, they describe a smell detecting tool "CODE AUDITOR", which checks source code for coding standards, *e.g.*, missing header block comments. In 1978 Krogh [28] not only demanded the presence of code comments, but also certain qualities of code comments: in the terminology of our paper, Krogh demanded comments of software *Purpose* and *Usage*, and also gave some examples of *Pointers*.

Since then, the research community studied a multitude of aspects of code comments. Some aspects are: the importance of code comments for readability, extensibility [50, 77], comment coherence [138], comment consistency [284], comment completeness [292], and comment adherence to coding guidelines [317, 360]. In the last decade, research on code comments often focuses on assessing the comment quality itself [91, 138], classifying comments automatically [198, 226], completing them [341], updating them [198, 333, 348], or even generating them [192, 240]. Such approaches often employ machine learning techniques, which mine code and comments from open source software projects, to create a learning database.

Our work employs a taxonomy for classifying class comments from Rani *et al.*, called Class Comment Type Model (CCTM). They employed their taxonomy on Smalltalk classes [320], but also gave a mapping of their taxonomy [319] to prior taxonomies used for Java and Python [226, 258]. In our work, we slightly adapt and extend the CCTM for our study set of Simulink and MATLAB projects. Kostić *et al.* give an overview of code comment taxonomies [332] and used a proposed taxonomy to classify multi-language comments [345]. However, their taxonomy is much more coarse-grained, than the CCTM.

Blasi *et al.* [311] studied comment duplication (Type I, III comment clones) in Java source code. They strived to identify problematic clones that were too generic or copy-pasted. We only searched for Type I comment clones in Simulink and MATLAB. Our classification also did not aim at finding problematic duplications, but at finding the duplication origin. We thus classified comment duplication as generic/copy-paste, library imports, IDE generation, or synthetic generation.

Simulink Comments

There has also been some prior interest in studying comments in Simulink. Pantelic *et al.* studied industrial Simulink projects and their evolution, as well as commenting practices [276, 314]. In [314], they studied the frequency of changes on various model comments during the model's development. They found that comments were least often changed. Within the comment changes, Annotations were changed most often, while DocBlocks remained mostly static. Pantelic *et al.* did not study the frequency of changes on Element Descriptions (block description, signal description) or Notes – we considered both features in our study. They also did not analyze the actual comment information or other characteristics like lengths or duplication. As they studied an industrial project, the experimental data and most basic information about the project itself is not available. In their anecdote-driven work [276], Pantelic *et al.* argue that current Simulink modeling practice faces several challenges: a lack of automation, (high quality) tools, and documentation templates. In fact, even a standard process of documentation is missing in a culture of prototype first, documentation third (or never). Pantelic *et al.* refute that (Simulink) models *are* already documentation, as the model only provides syntactical understanding, while documentation provides additional semantic understanding. They demand good documentation providing information about (1.) *software requirements specification*, which should give a model's black box behavior in a more abstract way than the direct implementation; and (2.) *software design description* (SDD), which should give semantics about the internal design, anticipated changes, hierarchy, and interfaces. The research group around Pantelic also developed a template for including SDD information into the model and a tool helping with the documentation process [249]. DocBlocks are created automatically, so that the developers can manually enter the documentation into a designated location. Their tool creates such DocBlocks for Purpose, Internal Design (focusing on interfaces), Rationale, and Anticipated Changes (see SDD, above). Developers are also expected to document changelogs and system acronyms/notation/definitions. Overall, their template covers the most-used categories of the CCTM used in our work.

While there are some studies, collecting open-source Simulink models [236, 337], and providing various metrics of models [1, 4, 356], none of those studies analyzed Simulink comments.

To the best of our knowledge, we are the first to study the commenting practice in open-source Simulink projects, as well as analyzing actual comment information of Simulink models. We are not aware of studies of comments in other visual modeling languages like UML or SysML.

3.4.8 Conclusion and Future Work

In this study, we found that open source MATLAB and Simulink projects feature a wide variety of types of comments, covering nearly the whole spectrum of the commentary taxonomy CCTM in addition to others. Many of the comments are duplicated by various means and are present in all levels of the model hierarchy, while developers focus mostly on the highest levels. We have shown that bigger and more complex models feature more comments and a higher total comment length, while each comment does not change in size. Model age on the other hand is neither a factor in model size nor comment amount. Finally, we found that the CCTM taxonomy is applicable for languages of different paradigms, and we extended it into a more complete taxonomy, named SCoT. We expect SCoT to be useful for classifying comments of all types, and languages, while probably needing only slight adjustments or additions.

We found comments in Simulink to only stand out in their many comment types in comparison to textual languages. In terms of information diversity and distribution, Simulink comments fall in line with all other studied languages. We proposed a number of ways to support developers in commenting their Simulink models. This could be done by modifying, or adding Simulink IDE features, greatly extending guidelines on Simulink comments, and comment smell detection – we expect many of our suggestions to also be useful for other visual languages and their tools.

While our work only learns from artifacts, the models and source code, in the future, we want to directly survey developers. Receiving opinions on how developers intend to document, their thought process while doing so, and their struggles, would put our findings into a more complete perspective. Similarly, we could scrape Simulink documentation related discussion from forums or mailing lists, like in [318], to gather insights into Simulink-specific documentation issues.

As the current guidelines on MATLAB and Simulink commentary are leaving many gaps and are not widely followed, we would like to create exhaustive modeling guidelines together with practitioners. This would be particularly useful in partnership with an industrial partner, as our current knowledge only comes from open source projects. After guideline synthesis, we plan to build a comment smell detector, which finds parts that need (more) commentary or even automatically refactors them.

4

Expanding Simulink Model Collections

This chapter addresses the two critical gaps in open-source Simulink datasets identified by our prior work (see Chapters 2 and 3): the lack of large-scale models and the lack of industry-like models. We present two complementary solutions in *GRANDSLAM: Linearly Scalable Model Synthesis* (Section 4.1), and *SMOKE2.0 Whitebox Anonymizing Intellectual Property in Models While Preserving Structure* (Section 4.2).

Synthesizer GRANDSLAM GRANDSLAM (Section 4.1) addresses the need for large-scale models by synthesizing large Simulink models with millions of elements. Unlike prior synthesizers, GRANDSLAM scales linearly, using building blocks mined from existing datasets like SLNET [337]. During its development and evaluation, we discovered and reported 11 confirmed bugs and 2 confirmed scalability issues in Simulink, demonstrating its dual role as a model generator and effective fuzzer. GRANDSLAM's capabilities enable researchers to conduct scalability experiments that were previously infeasible with smaller, open-source models.

Anonymizer SMOKE SMOKE (Section 4.2) addresses the scarcity of industry-like models by anonymizing proprietary Simulink models. Instead of attempting to synthesize industry-like models – a challenge due to the lack of clear, measurable, and standardized criteria beyond model size [208, 218, 1] – SMOKE preserves the realism of industrial models while removing sensitive information. This ensures researchers can access realistic models without violating intellectual property constraints, enabling industry-research collaborations.

Combining the Tools SMOKE and GRANDSLAM can also be combined. In the first step SMOKE would be used to anonymize a set of Simulink models. Next, GRANDSLAM can be used to mine building blocks from the anonymized models. In the last step, GRANDSLAM can then synthesize models from these custom, but anonymized building blocks.

Summary Together, these tools expand the scope of empirical research: GRANDSLAM supports large-scale experiments, while SMOKE bridges the gap between open-source and industrial models. They mitigate the limitations of existing datasets, enabling more robust and realistic studies.

↪ Thesis Roadmap

4.1 Grandslam: Linearly Scalable Model Synthesis

Authors: Alexander Boll.
Submitted to: 2026 IEEE 33rd International Conference on Software Analysis, Evolution and Reengineering (SANER).
Copyright: The author.
Abstract: Commercial cyber-physical system (CPS) development tools are complex and safety-critical software, yet face two major testing challenges. First, the lack of formal specifications necessitates fuzzing for bug discovery. Second, the scarcity of large, open-source models hampers effective scalability testing and empirical research. To address these challenges, we present Grandslam, a Simulink model synthesizer that scales linearly and generates models far larger than prior work. Grandslam enables both fuzzing and scalability testing at unprecedented scales, synthesizing diverse models with over 1M blocks. Our evaluation reveals eleven confirmed bugs and two confirmed scalability issues in Simulink, demonstrating its effectiveness.

4.1.1 Introduction

Commercial cyber-physical system (CPS) development tools, such as MathWorks MATLAB/Simulink [242], are fundamental platforms for designing, simulating, and analyzing CPS models for embedded code implementation. However, these intricate software systems are prone to bugs, posing significant challenges for developers, and risks in safety-critical domains. Detecting these bugs is challenging due to the lack of formal specifications and closed-source tools, so current research relies on fuzzing to uncover them [235, 291, 326, 364].

Yet, current state-of-the-art CPS fuzzers are limited to small and medium-sized models, while synthesis approaches struggle to generate valid models at scale [235, 325]. These limitations add to broader challenges in tool simulation model research, including reproducibility, replicability [369], and tool development. Large models are particularly critical for testing scalability of tools, and conducting empirical research, both of which rely on models of sufficient size and complexity [218, 267, 2, 5, 356]. Together, these gaps undermine both the reliability of CPS tools and the rigor of research aimed at advancing them.

To address these challenges, we develop a Simulink model synthesizer that (i) scales more effectively than prior approaches, enabling the synthesis of very large valid models, and (ii) supports the synthesis of models with various constraints, shapes, and sizes. Our synthesizer enables fuzzing and scalability testing, and also provides dataset augmentation for empirical research. We synthesize models larger than any currently available open-source models. While our primary focus in this work is on Simulink as a representative CPS development tool, we design our approach to be generalizable to other modeling and textual languages.

Our synthesis approach leverages our novel concept of *equivalence under substitution*. In a nutshell, our approach guarantees that a valid model remains valid if a model element is substituted. Given a dataset of models, we form equivalence classes, where a model element of a class can safely be substituted with any other element of its class. This ap-

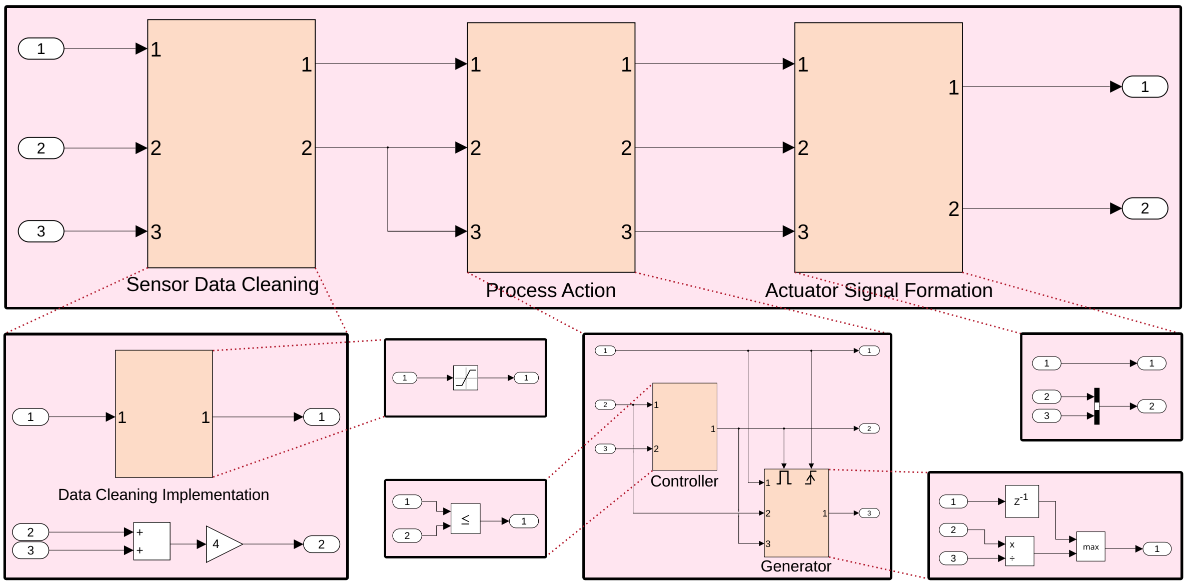


Figure 4.1. Complete Subsystem breakdown of a representative Simulink model.

proach enables us to implement our linearly scaling Simulink synthesizer GRANDSLAM (**G**iant **R**ecomposition **AND** **S**ynthesis of **LA**rge **M**odels) that synthesizes Simulink models by substituting Subsystems – Simulink's primary abstraction mechanism – that have equivalent interfaces. GRANDSLAM is written in MATLAB and comes with five highly configurable synthesis strategies, generating models of various shapes and sizes. In the evaluation of our approach, we found eleven confirmed bugs, and two confirmed scalability issues in Simulink. We further found missing elements in the Simulink Subsystem interface definition by Jaskolka *et al.* [294].

Overall, GRANDSLAM enables researchers and developers to systematically test and validate CPS tools at scale, improving both the reliability of the tools and the reproducibility of research. We summarize our contributions as follows:

- We develop a novel approach for synthesizing statically valid and/or compilable Simulink models.
- We implement our approach in a prototype GRANDSLAM with five strategies for synthesizing models of various shapes and sizes. To the best of our knowledge, this is the first model synthesizer that scales linearly.
- Our fuzzing revealed eleven confirmed bugs, and two confirmed scalability issues.
- We extend the definition of Simulink Subsystem interfaces from prior literature.

4.1.2 Background: Simulink

Simulink [242] is a block-diagram environment integrated with MATLAB and widely used for modeling, simulating, and analyzing dynamic systems across domains such as control engineering, signal processing, and embedded systems design. Its graphical interface enables intuitive representation of complex systems, while its tight integration with MATLAB facilitates algorithm development, data analysis, and automated code generation for real-time applications.

Simulink models are block diagrams composed of Blocks and Lines as their core model elements. Blocks are connected by Lines and Lines carry information from Block Outports to Block Inports during model simulation. Lines can hold values of different types (*e.g.*, boolean, double, int16) and dimensionality (*e.g.*, scalars, 1-D vectors, matrices). Meanwhile, Blocks transform their input from Inports to output at their Outports. To handle complex models, developers can use Subsystem Blocks and use them to hierarchize the model. A model's root Subsystem and its content is given at the top of Figure 4.1. We show all Subsystems' internals at once (normally only one is visible at a time), *e.g.*, the Subsystem 'Sensor Data Cleaning' is presented on the bottom left of Figure 4.1. 'Sensor Data Cleaning' contains the nested Subsystem 'Data Cleaning Implementation'. In addition to normal Inports and Outports, Subsystems may also have a SpecialPort. SpecialPorts in Simulink are: EnablePort, TriggerPort, ActionPort, PMIOPort, or ResetPort. There is an Enable- and a TriggerPort on top of the Subsystem 'Generator' in Figure 4.1. A Subsystem's closest analog in textual languages are functions, or methods. While the figure contains several other model elements, only the following ideas are essential for the subsequent understanding: models may have (nested) Subsystems and each Subsystem has a number of typed Ports connecting them with other model elements.

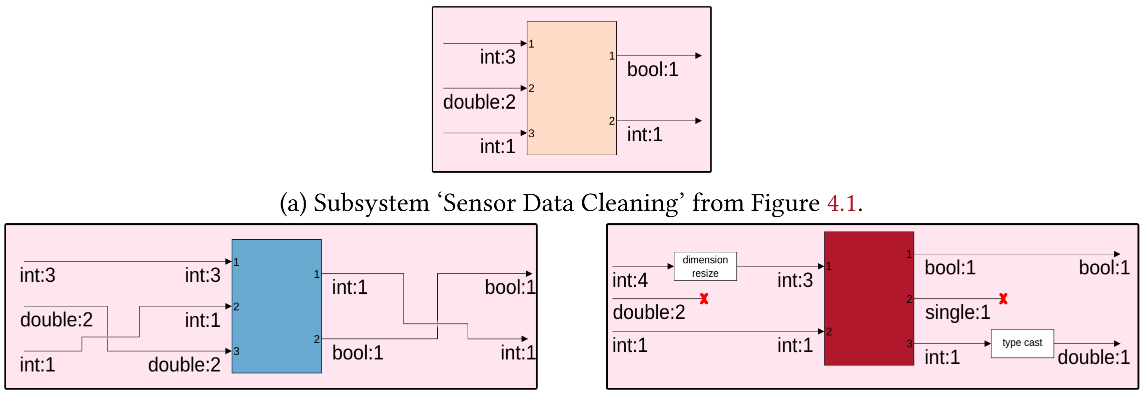
Simulink models adhere to multiple levels of *validity* [235, 308]. In this work, we focus on models being *statically valid*, *compilable*, and *simulable* (the model can be simulated, *i.e.*, it runs without runtime errors.). Each level presupposes the prior one. Users can verify whether a model is statically valid or compilable by loading, and compiling it. As we will show in Section 4.1.5, most compilable models are also simulable.

4.1.3 Subsystem Equivalence under Substitution

Definition of Subsystem Equivalence

Our main idea in this paper is to synthesize models by substituting Subsystems. A *substitution*, is a model transformation that removes a Subsystem S_o (original) and replaces it with a Subsystem S_s (substitute) exactly in its place, including all prior Line connections. A substitution does not require the two Subsystems to be from two different models, *i.e.*, the most trivial substitution is to substitute a Subsystem with itself ($S_o = S_s$). To ensure meaningful substitutions, we restrict S_s to those that maintain the model's validity after substitution. Specifically, we require the resulting model to retain *static validity* and *compilability*. Unlike textual programming languages, models can provide value even if they are not compilable, as long as they are statically valid [1]. By supporting multiple levels of validity, our approach enables fuzzing and scalability testing for a wider range of Simulink models and tools. As the metamodel of Simulink isn't published, we conjecture substitution validity based on the official Simulink documentation and Jaskolka *et al.*'s Subsystem interface definition [294]. Our later experiments strengthen and extend their interface definition and our conjectures.

We can ensure a model remains statically valid, if after the substitution S_s is connected exactly at the same Lines as S_o. This gives our first definition of *equivalence under substitution*,



(a) Subsystem 'Sensor Data Cleaning' from Figure 4.1.

(b) An adapted order-relaxed-equivalent Subsystem.

(c) An adapted Subsystem under number-, type-, and dimensionality-relaxation.

Figure 4.2. For substitutions, adaptations may be required, such as rewiring, dangling lines, type casting, or resizing dimensionality.

or *untyped-equivalence*: S_o and S_s must have the same number of Lines at their Inports, SpecialPorts, and Outports. E.g., the Subsystems 'Sensor Data Cleaning' and 'Actuator Signal Formation' in Figure 4.1 are untyped-equivalent; all other Subsystems of the model have pairwise different numbers of connected Inports and Outports and are not substitutable. With this definition, a substitution becomes easy: we remove a Subsystem, and replace it by connecting the old Lines with the Ports of the new Subsystem.

For the second equivalence level, we aim to ensure compilability. During compilation, Simulink checks the types and dimensions of Lines: the type and dimensionality from every Outport must conform (in general be identical) to the type that is expected at the other end of the Line (an Inport). This inspires our second definition of *typed-equivalence*: S_o and S_s must be untyped-equivalent, with all connected Lines having identical types and dimensions. Note that this definition is very conservative. During our automatic inspection, we can only observe that a Port *in its current context* supports, *e.g.*, int32. It may also support, *e.g.*, int16, or int8, though.

Equivalence Classes and Relaxed Equivalence

All Subsystems that are equivalent to each other fall into (typed or untyped) *equivalence classes*, *i.e.*, every Subsystem is substitutable by all members of its equivalence class. Each class is defined by inputs and outputs, *i.e.*, the 'interface' of the representative of its class.

With our goal of substitution in mind, equivalence classes should not be singletons, *i.e.*, sets with one element, in order for substitution to be possible. To reduce the number of singletons, we define four variations of *relaxation*, that follow the common adapter patterns for harmonizing function calls from classical programming languages. In *order-relaxation* (argument reordering), we allow for *rewiring* of Lines during substitution: the order of the Inports and Outports, and their types or dimensionality may vary, as long as there is a match for each of them. E.g., in Figure 4.2a for each input and output Port there is an equivalent input and output Port in Figure 4.2b; inputs 2, and 3 and the outputs need to be rewired. In *type-relaxation* (parameter coercion and return value mapping), we allow

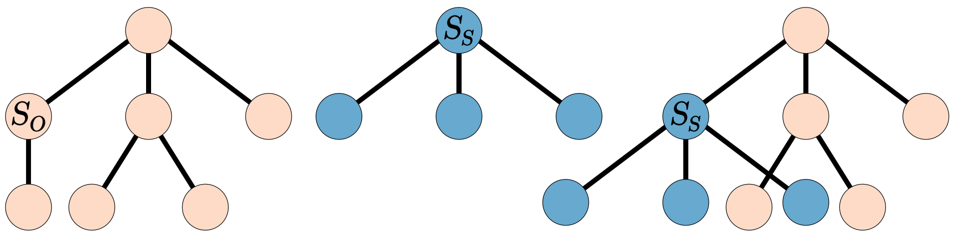


Figure 4.3. S_o is substituted by the equivalent S_s from another model, resulting in the larger model shown on the right.

for the conversion of data types that can be safely cast, *e.g.*, `int16` to `int32`. Lastly, we define *number-relaxation* (argument truncation and return value suppression): a Subsystem is *number-relaxed* to another if it uses a subset of its inputs and produces a superset of its outputs; *dimensionality-relaxation* can be defined similarly. The concepts of *type-relaxation*, *number-relaxation*, and *dimensionality-relaxation* are shown in Figure 4.2c.

Impact of Substitution

In Figure 4.3, we depict the impact of substitutions on a model. On the left, we can see the complete Subsystem hierarchy, or *Subsystems tree* of the model from Figure 4.1. If the Subsystem tree of S_s is bigger than the one of S_o, the model will grow when their Subsystem trees are substituted. This observation is key to our GIANT synthesis strategy (see Section 4.1.4), where we iteratively apply such substitutions.

4.1.4 Implementing our Approach in Grandslam

To implement our approach, we first mine a given dataset of models to construct two hash tables: `interface2ids`, which uses a description of a Subsystem's interface as a key to look up its equivalence class, and `id2subinfo`, which is used to look up properties of Subsystems. In the later synthesis step, we use these hash tables extensively. We thus first present the mining and their construction.

Encoding and Mining Equivalence Classes

The mining of equivalence classes iterates over every model's Subsystem and each of its Inports, Outports, and SpecialPorts. We encode the hash-key of a Subsystem interface as a string. For untyped equivalence, we quantify their number, such as '`3,2`' for 3 Inports, 2 Outports, and no SpecialPort.

For typed equivalence, we have to differentiate Port type and dimensionality. For each Port, we denote its type and dimensionality as '`<type><dim1>+...+<dimN>`'. The added '+' differentiates a 2x2 matrix from a 22-row vector. For SpecialPorts, we denote their name. Next, an Inport, Outport, and SpecialPort hash is formed by joining their individual Port hashes with a '`:`'. Finally, the overall interface hash is formed by joining the three partial hashes with a '`,`'. Some examples: The hashes of the Subsystems in Figure 4.2 are '`int3:double2:int1,bool1:int1`' (Figure 4.2a), '`int3:int1:double2, int1:bool1`' (Figure 4.2b), and '`int3:int1,bool1:single1:int1`' (Figure 4.2c).

The values of `interface2ids` are all the *identifiers* of Subsystems that belong to the equivalence class of its interface hash-key. We construct an identifier that describes the absolute position of the Subsystem by combining: the Simulink model file it belongs to, the relative path of the Subsystem within the model, and the Subsystem's name. The identifier of 'Controller' in Figure 4.1 may thus be: `"ExampleModel/Process Action/Controller"`. Given an interface of a Subsystem, one can thus quickly lookup equivalent substitute candidates in `interface2ids`.

To mitigate filling the equivalence classes with clones, *e.g.*, from libraries, and thus substituting a Subsystem with itself, we deduplicate each equivalence class. We classify Subsystems from the same equivalence class as duplicates, when they have the same number of direct children, number of Subsystem children, and local Subsystem tree depth. These are metrics we compute for our synthesis strategies, and suffice as a quick heuristic for our prototype. However, the heuristic does not guarantee that every Subsystem of an equivalence class is truly unique.

Encoding Relaxation

We encode order-relaxation by *normalizing* the interface hashes. This can be achieved, by sorting the Inport, Outport, and SpecialPort hashes alphabetically, before joining them. Both Subsystems in Figures 4.2a and 4.2b get the hash '`double2:int1:int3,bool1:int1`' proving they are order-relaxed-equivalent. The other three relaxations could be achieved by adding a Subsystem not only to its direct equivalence class, but also to every other equivalence class, that is more specific. We only followed up to implement order-relaxation in our prototype.

Mining Subsystem Properties

We use the hash table `id2subinfo` to look up various properties of Subsystems. The previously introduced identifiers are the keys of `id2subinfo`. The most valuable properties of a Subsystem stored in `id2subinfo` are the identifiers of all direct child-Subsystems as well as its own interface which enables the lookup of the Subsystem's equivalence class via `interface2ids`. Depending on synthesis goals, one can also store other properties. We store the Subsystem's own depth within the model, the depth of its local Subsystem tree, and the number of its local Blocks.

Synthesizing and Synthesis Strategies

Using the combination of `id2subinfo` and `interface2ids`, we can now elegantly synthesize models using two paradigms:

1. Synthesizing from the ground up: This paradigm starts by synthesizing a new model M using a random root Subsystem r from the dataset as the seed. We substitute each of r's Subsystem children, by another Subsystem from its respective equivalence class, if a suitability criterion for the substitution is passed. Depending on relaxation, adaptions may have to be made for a valid substitution (see Figures 4.2b and 4.2c). This process is then repeated until the Subsystem tree is completed, *i.e.*, all leaf Subsystems are processed. The pseudocode of our approach is given in Listing 4.1.

We define four suitability criteria (*i.e.*, synthesis strategies) within this paradigm: Random: any Subsystem from the equivalence class passes. Role: given a separate 'role model' M'

```
1  Input: Subsystem tree rooted in r
2  Output: The synthesized model M
3
4  //Init model M and Stack S of TODO Subsystems
5  M = r;
6  S = id2subinfo(r).childSubsystems;
7  while S ≠ ∅ do
8      s = pop(S);
9      //Lookup equivalent Subsystems
10     E = interface2ids(id2subinfo(s).interface);
11     //Choose k elements e from E as substitute candidates; avoids
         uniform substitutions
12     E' = sample(E, k);
13     for e ∈ E' do
14         if isSuitable(s, e, M, E') then
15             M = substitute(s, e, M);
16             s = e;
17             break;
18         endif
19     endfor
20     //Repeat for substitute's children
21     S = S.append(id2subinfo(s).childSubsystems);
22 endwhile
```

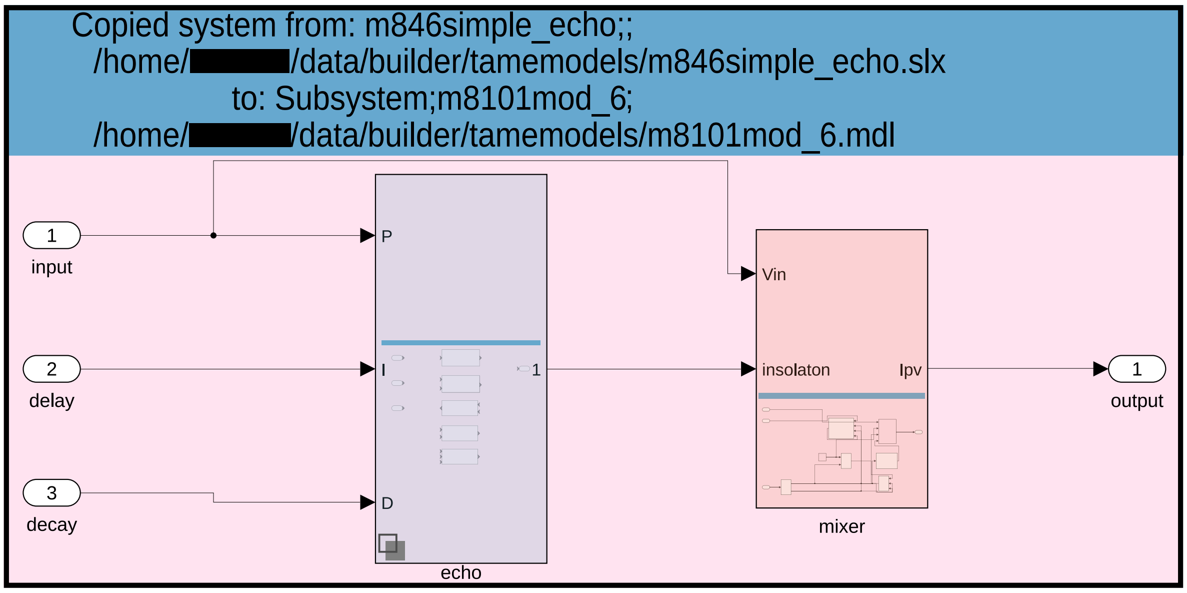
Listing 4.1. Pseudo-code of synthesis from the ground up.

from the dataset, we synthesize M with an isomorphic Subsystem tree to M'. A substitute Subsystem is thus suitable,[1] when it has the same number of children as the role model's Subsystem has at the same position. Bushy: From a sample from the equivalence class, the Subsystem with the most children is suitable. Deep: From a sample of Subsystems from the equivalence class, the one with the biggest Subsystem tree depth is suitable.
We propose Random and Role to synthesize models that are similar in size to the original models, and Bushy and Deep to synthesize large models. We expect Bushy to create wide and dense Subsystem trees, and Deep to create deep Subsystem trees.

Without further moderation, our experiments showed that Bushy and Deep tend to grow infinitely. We thus introduce a backtracking mechanism that activates, when a previously selected maximum depth of the model is overshot. The backtracking voids substitutions in shallower levels and tries other candidates instead, until the complete Subsystem tree obeys our constraints. A similar backtracking also activates for Role, when no local candidate Subsystems can be found to keep the local Subsystem trees isomorphic.

Note, that while we choose root Subsystems as the seed of our models, any other Subsystem also works. We focus on root Subsystems as seeds because the dataset models are rooted by them by definition. However, if the chosen root Subsystem has no children, the synthesis process terminates immediately. In such cases, another root must be tried, except if M' in Role is also just a singular Subsystem model without children.

[1] isSuitable in line 14 of Listing 4.1 needs M' as an additional parameter.



Copied system from: m846simple_echo;;
/home/[redacted]/data/builder/tamemodels/m846simple_echo.slx
to: Subsystem;m8101mod_6;
/home/[redacted]/data/builder/tamemodels/m8101mod_6.mdl

Figure 4.4. Substituted Subsystems show our annotations in blue, including in the small Subsystem previews.

The big models constructed by Bushy and Deep are assembled from hundreds or even thousands of models. Since Simulink's model loading, and Subsystem substitution are computationally expensive – and must be executed hundreds or thousands of times – we also developed an alternative paradigm for large models, based on fewer unique models, and fewer substitutions:

2. Recomposing existing models: The second paradigm recomposes existing models, embodying our fifth and final strategy: Giant. Instead of starting the synthesis with a single root Subsystem – for our last strategy Giant, we use a complete model as its seed. In every step, we select a partial Subsystem tree from anywhere in the model and try to substitute it with an equivalent Subsystem tree (see Figure 4.3), when it is suitable. To quickly grow a model, we found that alternating between two suitability criteria at each depth level improves the synthesis process of Giant: substitute Subsystem has deeper tree depth, and substitute Subsystem has more direct children. To create large models, we let a model grow with Giant, until the recomposed model trumps the biggest models from the dataset in their maximum number of model elements, maximum depth, and maximum number of Subsystems all at once. Giant can be stopped at any point, as after each substitution in Giant the model's Subsystem tree is complete. Stopping Deep or Bushy, however, needs a phase of capping off all unprocessed Subsystem trees with leaves. The other main difference between them is that Giant recombines existing models, where large parts of the Subsystem trees remain intact. Bushy and Deep create a patchwork of Subsystems from various models, because each node of the Subsystem tree is chosen individually.

In our implementation, we decoupled the building of a *blueprint* of the model and the actual *construction* of the model with Simulink API calls. This enables 'completely dry builds', where the Subsystem tree outline is completed but no actual Simulink model is created yet. The construction phase of Giant copies Subsystem trees (see Figure 4.3), the others

Table 4.1. Summary of the SLNET dataset.

	SLNET models	**Stable**	**Compilable**	Simulable
Models	9,095	**5,912**	**761**	697
Subsystems	N/A	**313,820**	**1,807**	N/A

synthesize their Simulink models one model element at a time. In the construction phase, we add annotations to every Subsystem denoting their provenance, (see Figure 4.4).

4.1.5 Methods

Dataset Mining

In order to construct our hash tables `interface2ids` and `id2subinfo`, as described in Section 4.1.4, we mine the SLNET data set by Shrestha *et al.* [337]. SLNET is a diverse and large dataset of 9,095 Simulink models stemming from 2,837 Simulink repositories. The set encompasses various domains, model sizes, modeling goals, and has become a benchmark in the field as it has been used in a number of empirical studies [10, 5, 356, 11, 6, 372, 373]. Before mining the Subsystem equivalence classes, we filter the SLNET model set for *stable* models: each model should be statically valid, *i.e.*, free of errors while loading, it should also not cause any Simulink bugs, while loading, compiling, simulating, saving and closing. We compile every stable model and simulate it for 10 seconds to assess both compilability and simulability. We do this by performing all the mentioned model operations and restarting the script, whenever Simulink crashes (hard crashes or stops without a catchable exception). This filtering step is vital for our later scripts as they have to handle thousands of models automatically. Especially the synthesis of large models becomes brittle if just a few models in the dataset are unstable. To further 'tame' the models, we remove all Callbacks to mitigate side effects (see our results of **RQ2.3**). After filtering our dataset, it contains 5,912 models that we mine for untyped-equivalence and 761 models for typed-equivalence – both in bold in Table 4.1.

Research Questions

As we intend to compose models by substituting equivalent Subsystems, we first evaluate:
RQ1: Is our approach of repetitive substitution feasible? Our substitution approach assumes a small set of equivalence classes, ensuring that a significant portion of Subsystems can be replaced by equivalent alternatives. Untyped equivalence might seem to limit the equivalence classes – a maximum of, *e.g.*, 10 Inports and Outports yield just 121 combinations – but SpecialPorts can inflate this number considerably. For typed-equivalence, it may be possible that most Subsystems are fragmented into singletons classes, as inputs could vary by dimension and type. Additionally, not every model is compilable, and thus the number of Subsystems that are available for our analysis is lower from the start. If too many equivalence classes are singletons, then we relax our equivalence requirements (see Section 4.1.3).

We further evaluate the feasibility of our proposed strategies (see Section 4.1.4). We run all five strategies with both typed and untyped equivalence, and measure how they perform in regard to success rate and scalability. For RQ1, we answer the following sub-questions:

RQ1.1 How are the equivalence classes constituted?

RQ1.2 Can we synthesize models with untyped and typed equivalence classes?

RQ1.3 How do the blueprint and construction phases scale?

Once we are satisfied with the general feasibility of our approach, we investigate the synthesized models: **RQ2: What are the synthesized models' characteristics?** Our goal in RQ2 is to gauge the synthesized models broadly, before we use them for fuzzing or scalability tests. We measure basic model metrics [1, 5] to answer the following sub-questions:

RQ2.1 How large are the synthesized models?

RQ2.2 How diverse are the models?

RQ2.3 How many models are compilable, and simulable?

Lastly, we evaluate the suitability for our use cases:
RQ3: Are the synthesized models suitable for fuzzing and scalability tests? For fuzzing, we analyze all program halts encountered during synthesis and analysis. Most halts stem from implementation bugs in our scripts (which we fuzz 'along the way'), while others reveal Simulink bugs, which we report to MathWorks. Additionally, we identify a few bugs through ad-hoc testing. Related fuzzing approaches are discussed in Section 4.1.8, as they primarily target compiler bugs, whereas our focus is on a broader range of model actions.

To assess scalability, we reuse the synthesized models from RQ2 and apply various model actions to them. By iteratively computing regression curves while relaxing model size constraints, we determine whether larger models provide deeper insights into scalability behavior. Comparisons with related approaches are deferred to Section 4.1.8, as their synthesized Simulink models are limited in size and thus less informative for scalability analysis.

4.1.6 Results

RQ1: Is our approach of repetitive substitution feasible?

RQ1.1 How are the equivalence classes constituted?

Table 4.2 gives the main results for our first research question. One can see the strong impact of deduplication, that removes Subsystem clones – both intra- and inter-model clones – from an equivalence class. The drop in size of the equivalence classes via deduplication can also be seen in Figure 4.5. Since substituting a Subsystem with a clone of itself serves no purpose, and the two typed configurations are nearly identical (order-relaxing unifies only 3 classes), we move on to analyze only the configurations highlighted in bold in Table 4.2 and Figure 4.5.

42% of the untyped equivalence classes are singletons, which means the equivalence class' representative has a unique interface, and we cannot find a substitute. This number goes up to 72% for the typed equivalence classes. Despite the prevalence of singletons, the data in the final two columns of Table 4.2 demonstrates that a randomly selected typed Subsystem retains substantial substitutability – with an average of 40 distinct replacement possibilities from 39 unique models.

Table 4.2. Equivalence classes across configurations.

	#classes	singleton class %	mean other Subsystems	mean other models
untyped				
with duplicates	326	20	49,582	1,927
unique	**326**	**42**	**532**	**508**
typed, strict				
with duplicates	202	46	273	249
unique	202	74	40	38
typed, order-relaxed				
with duplicates	199	44	273	249
unique	**199**	**72**	**40**	**39**

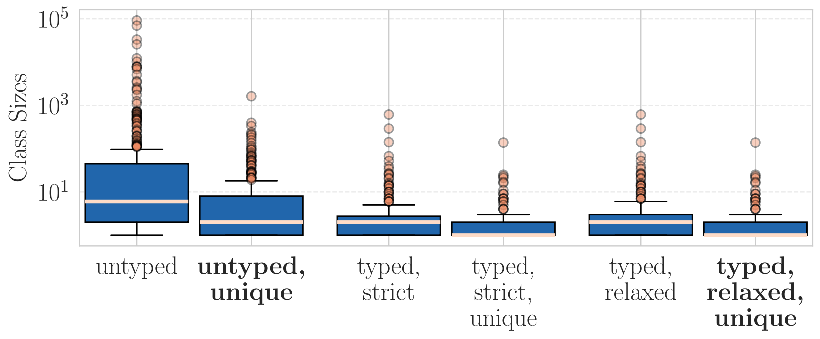
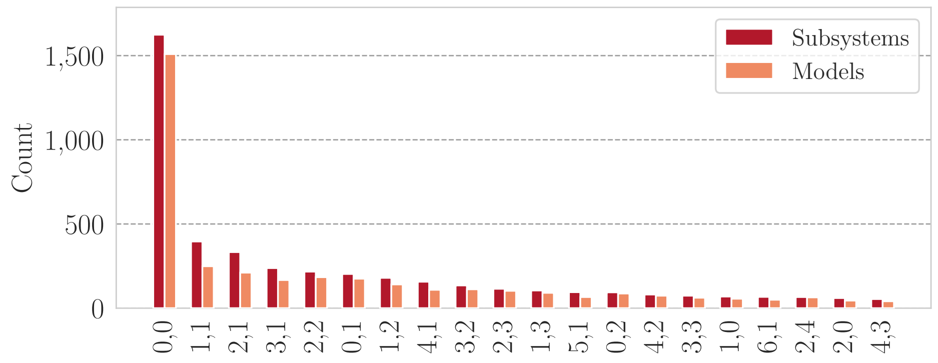


Figure 4.5. Box plots of equivalence class sizes.

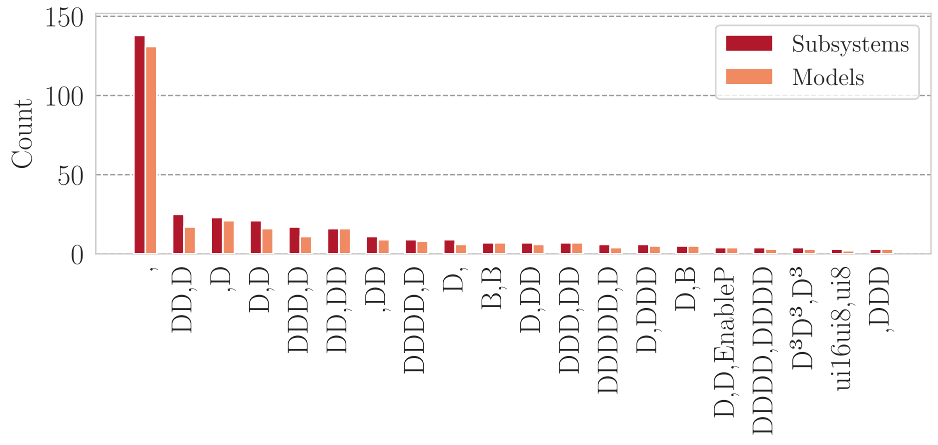
Our analysis of the most frequent equivalence classes and their interfaces (see Figure 4.6) reveals that the most common classes have few Inports and Outports, and seldomly feature SpecialPorts. By far the most common Line type is uni-dimensional `double`, but we also found `boolean`, `uint8`, `uint16`, and three-dimensional `double` in the top 20 interfaces. Furthermore, a class' Subsystems are spread over many models and not just populated by a single model's Subsystems, noticeable by the similar heights of the peach and dark peach bars in Figure 4.6.

RQ1.2 Can we synthesize models with untyped and typed equivalence classes?
We ran each of our five strategies once with the untyped, unique equivalence classes, and once with the typed, relaxed, unique equivalence classes (bold in Table 4.2 and Figure 4.5). With Random and Role we generated 1,000 models, with the others 100 models, totalling 4,600 models over all runs. Every synthesized model is a statically valid Simulink model and can be opened and edited in Simulink.

While all strategies generated their quota, we noticed bigger failure rates (see first row in Table 4.3) for generating typed models with Bushy and Deep. In the blueprint phase, the high number of singletons leads to a high number of backtracking, until a seed is given up (we allow for 3 tries per Subsystem tree node). Giant can skip over the bottlenecks that trouble



(a) Interface hashes of untyped, unique equivalence classes.



(b) Interface hashes of typed, relaxed, unique equivalence classes.

Figure 4.6. Distribution of the 20 most frequent equivalence classes and their occurrences in Subsystems and Models.

Bushy and Deep by substituting larger Subsystem trees instead of single Subsystems. Our construction phase failed in only 0.8% of attempts, resulting from bugs in our construction implementation.

RQ1.3 How do the blueprint and construction phases scale?

The first two rows of Figure 4.7 show that the blueprint phase (1,000-10,000 Blocks/s for the large strategies) is much faster than the construction phase (100-1,000 Blocks/s). The linear relationship for untyped Bushy between the times in blueprint and construction phase vs. the number of elements is clearly visible in Figure 4.8. The other methods also scale effectively linear, though with less clarity. We highlight Bushy because it generated the largest models by a significant margin.

RQ2: What are the synthesized models' characteristics?

RQ2.1 How large are the synthesized models?

The third and fourth rows of Figure 4.7 show that Random and Role generate models with similar sizes to their SLNET datasets. Bushy, Deep, and Giant's models are much larger. The untyped Giant models all have more than 108,523 Blocks (typed more than 1,122) – the maximum size in SLNET. The largest model of Bushy comprises more than 1.1 million

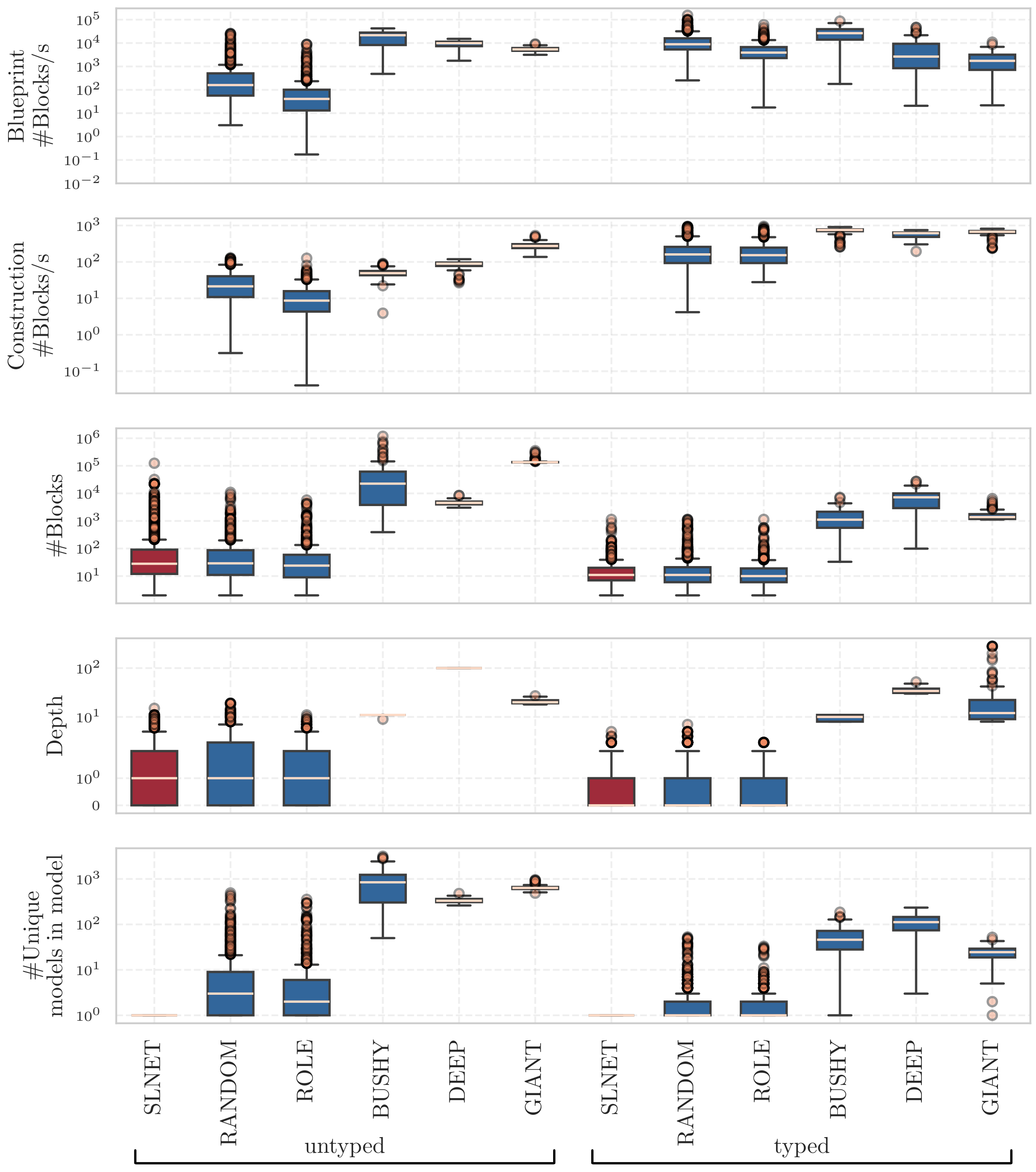


Figure 4.7. Box plot comparison of SLNET and our synthesis strategies. All plots are scaled logarithmically, except ‘Depth‘ which is scaled symmetrically logarithmic [121]. All times are measured on an MX Linux laptop with an i9-13980HX CPU, 94GB RAM, and MATLABR2025a.

Table 4.3. Various metrics across our strategies.

	Random	Role	Bushy	Deep	Giant
Blueprint failure rates					
untyped	.000	.079	.099	.174	.007
typed	.000	.040	.888	.996	.145
mean pairwise Jaccard similarity (models)					
untyped	.003	.002	.265	.249	.398
typed	.002	.002	.178	.309	.290
mean pairwise Jaccard similarity (Subsystems)					
untyped	0	0	.007	.001	.058
typed	.001	.001	.076	.023	.201
% of SLNET models covered					
untyped	48.3	44.1	79.5	50.2	47.9
typed	87.9	81.2	56.4	61.9	23.5
% of SLNET Subsystems covered					
untyped	4.8	3.7	79.1	15.1	69.6
typed	78.1	60.1	69.9	75.8	52.2
% compilable					
untyped	14.5	13.6	0	0	0
typed	94.1	96.3	19	23	24
% simulable					
untyped	13.9	12.9	0	0	0
typed	89.9	92.9	15	23	15

Blocks. The typed models exhibit a reduction in size by an order of magnitude or greater. For untyped models, our depth constraints (8-12 for Bushy, 50-99 for Deep) were nearly always maxed out. We lowered these constraints for typed Deep (30-50), otherwise the failure rates were impractically high.

RQ2.2 How diverse are the models?

The final row in Figure 4.7 shows the number of models whose Subsystems were recomposed into each new model. For example, Giant recomposed Subsystems from approximately 600–1,000 different models per synthesized model. The large strategies lead to very high diversities with more than 100 models for typed and more than 10 models for untyped classes. Judging the mean pairwise Jaccard similarities [331] of models and Subsystems (see row three and four in Table 4.3), we find all strategies to be highly dissimilar. Table 4.3 further gives the percentages of SLNET models/Subsystems that were covered by a strategy's synthesized models, *e.g.*, the 100 untyped Bushy models used nearly 80% of the models and Subsystems from the SLNET dataset for their synthesis.

RQ2.3 How many models are compilable, and simulable?

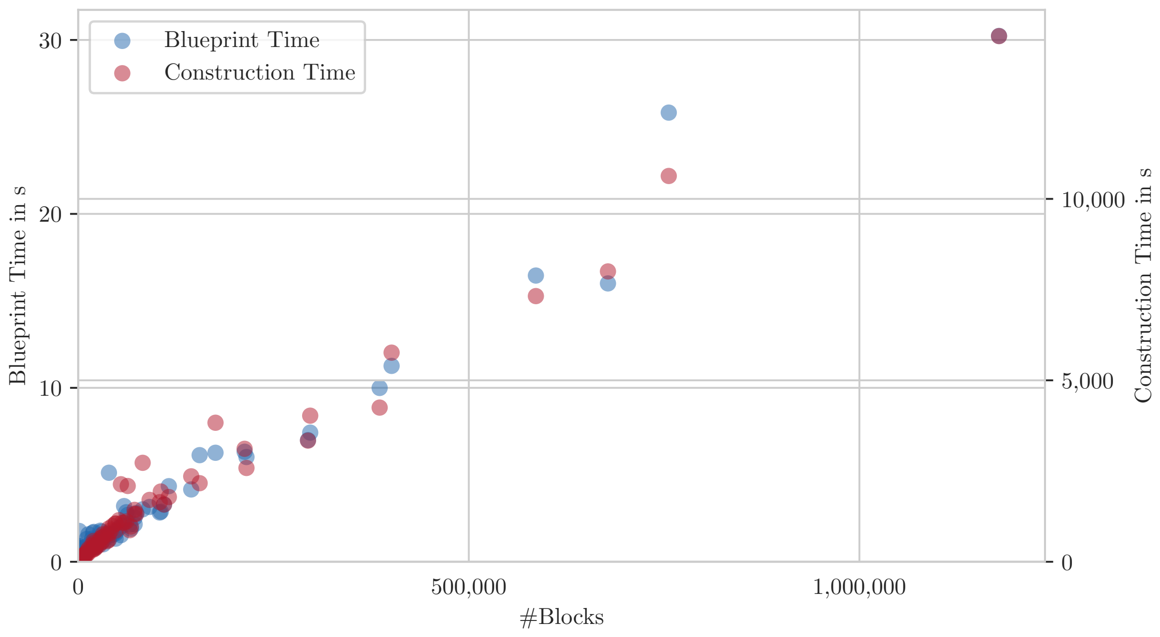


Figure 4.8. The linear scaling of untyped Bushy.

The last two rows of Table 4.3 show that the typed models have a much higher chance of being compilable, and the large strategies rarely produce compilable or simulable models. We found the low number of compilable typed models to be the result of an incomplete interface definition. In our definition of equivalence in Section 4.1.3, we followed the manually crafted interface definition by Jaskolka *et al.* [294].

In our practical experiments, we found three extensions to the definition of Jaskolka *et al.*, where some model elements can have a 'spill-over' effect, *i.e.*, a single Subsystem can dictate properties of other Subsystems globally: (1) Callbacks: models, Blocks, and Ports may have Callbacks, that are called and executed by various events. These events can be: the model is loaded, closed, it is saved, a Block is renamed, the model simulation starts, etc. These Callbacks are common MATLAB scripts, and they can – in rare cases – rename, or delete other model elements, which may *dynamically change a Subsystem's interface* and thus equivalence class. When synthesizing models with thousands of Subsystems, a single bad Callback can break the whole model. We thus repeated all of our experiments with the previously described 'taming' of the dataset. (2) Asynchronous Blocks: Certain Blocks in Simulink require the entire model, including all other Subsystems, to operate under asynchronous semantics. Thus, each Subsystem's interface needs to be extended with a label 'synchronous' or 'asynchronous'. (3) Locality of type and dimension declaration: Signal types and dimensions can be the result of (i) manual declaration at some point in the model, (ii) implicit generation from a model element that can only generate such type and dimension, (iii) inheriting the type and dimension from either (i) or (ii) from the prior Signal path. Thus, our equivalence definition for each type and Signal needs to be extended with labels (i)-(iii). During the synthesis, one then needs to order Subsystems from Signal sources to sinks (source-sink ordered depth first traversal), to guarantee valid type and dimension

Table 4.4. Summary of confirmed bugs.

Case Number	Description of Bug
Discovered during ‘taming’	
07130317	Hard crash in `set_param`
07131902	Hard crash in `set_param`
07144514	Hard crash in `set_param`
08020574	Message box flood in `compile`
08020724	Irrecoverable exception in `load_system`
Discovered during synthesis experiments	
06653446	Loaded model throws exceptions, errors
08112233	Irrecoverable exceptions in `compile`
08128597	Hard crash in `compile`
08147973	RAM usurped in `close_system`
08160317	Hard crash in `findClones`
08160318	Hard crash in `findClones`

propagation in the synthesized model. Cross-dependent type propagations, where Subsystem A needs B, and B needs A for a Signal’s type, are not resolvable with this concept though.

For the simulability, we only found divisions by zero, or algebraic loops to be causing halts – both of which our equivalence approach is not intended to catch.

RQ3: Are the synthesized models suitable for fuzzing and scalability tests?
In our experiments, we found eleven confirmed bugs (see Table 4.4). In addition, we found five bugs (two inconsistent outputs during ‘taming’; two memory leaks and a failed assertion with synthesized models), which we experienced multiple times but could not consistently reproduce.

For the scalability tests, we computed regression curves for the model actions in Table 4.5. Only ‘finding clones in a model’ and ‘closing a model’ showed quadratic scaling. For ‘finding clones’, the quadratic regression started to be the best fit at around 10,000 elements – for ‘closing models’ very large models with more than one million elements had to be included. The coefficient of determination R^2 [37] in the last column shows that all regression curves are strong fits. Because of bug 08147973, we could not perform an automated RAM regression analysis. Our preliminary manual analysis suggests that Simulink allocates roughly 2,900 bytes of RAM per model element, scaling linearly. After we sent MathWorks our largest model and confronted them with our scalability analysis of its loading time and RAM allocation, they confirmed two scalability issues and plan to fix them in future releases.

4.1.7 Discussion

Suitability and versatility of our approach

All five Grandslam strategies performed well and created models with properties we aimed for. Even with the high number of singleton equivalence classes in the typed setting, all five strategies succeeded in synthesizing models, though failure rates increased significantly,

Table 4.5. Scalability regression curves $Time = an^2 + bn$ and $Space = bn$, with n = number of model elements.

	a (quadratic)	b (linear)	R^2
Time in microseconds			
Load Model	—	30.947	.98
Find Elements	—	.242	.81
Save Model	—	27.610	.95
Find Clones	.007,978,387	−178.301	.95
Close Model	.000,004,412	7.603	.93
Space in Bytes			
File Size	—	122.811	.98

even for smaller models. We thus suspect that our typed setting approached the upper limit of singleton equivalence classes for which our strategies remain effective. The untyped setting was far more permissive, and it was there that we synthesized the largest statically valid models – by a substantial margin – of which we are aware. The largest statically valid model was generated by untyped Bushy and has more than 4.3M model elements, 1.1M Blocks, 110k Subsystems, 80k of which are unique and sourced from more than 3k unique models. This is about 400 times larger than synthesized Simulink models in prior work (see Section 4.1.8), and much larger than typical industrial models that seldom have more than 20k Blocks [1].

While every model we generated was statically valid, the success rate of creating dynamically valid models was much lower (see **RQ2.3**). Still, 66 of the 300 larger typed models we generated are compilable. The largest models that are compilable/simulable were generated by typed Giant and encompass 1,660/1,419 Blocks, 148/148 Subsystems stemming from 16/12 unique models.

Our five strategies already demonstrate the versatility of our approach in the resulting models and the process, *i.e.*, by dynamically switching the `isSuitable`-function based on the current synthesis status. Completely different strategies are easily integrated, with our approach: if one, *e.g.*, desires to synthesize models with many Blocks of type t, one can extend the `id2subinfo` in Section 4.1.4 with the count of t. For speed and effectivity of `sample`, one can also sort each equivalence class in `interface2ids` by number of t.

In this work, we tried five strategies in different settings, but left a number of variation points unexplored. (1) While order-relaxation has nearly no impact at all (Section 4.1.6), number-, type-, and dimensionality-relaxation may offer a significant reduction of singleton classes for the typed setting. (2) The deduplication of equivalence classes may be sharpened with a better heuristic with less false negatives and false positives. (3) Our strategies always choose root Subsystems as their root Subsystem, but Subsystems from any level could be a model's root – the synthesized models may be smaller, on average though. (4) Furthermore, our strategies have many more tunable parameters, which we chose by intuition and did not systematically explore, *e.g.*, the sample sizes k, minimum/maximum depth of Subsystem trees, etc.

Fuzzing and scalability potential

While our approach uncovered eleven bugs, only one (case 08147973) was directly attributable to large model size. Reproducing bugs with large models proved complex, if it required replaying time-consuming synthesis sequences (see Figure 4.8). In fact, MathWorks declined to fix one reported issue due to its complex reproducibility, highlighting a broader challenge in fuzzing: the value of bugs triggered by nonsensical or overly complex inputs [99, 108, 143]. This suggests that synthesizing many smaller models may be a more efficient fuzzing strategy.

In contrast, we view large models as excellent subjects for scalability testing, as their performance provides insights into how their underlying dataset – from which they were synthesized – would behave at scale. Our scaling experiments revealed a surprising asymmetry in core tool performance: while opening a model scales linearly, closing it exhibits quadratic scaling. MathWorks plans to fix two performance issues in relation to our scalability findings.

Generalizing to other elements and languages

As we have seen, our synthesis approach of substituting equivalent *Subsystems* works very well in Simulink. More general though, any combination of model elements with a definable interface and an equivalence relation is suitable for our substitution approach. This means individual Blocks or groups of connected Blocks could be used to build equivalence classes. We chose Subsystems, as they naturally limit the number of interfaces we have to analyze, and are the natural abstraction unit in Simulink. This means, that our approach can be generalized to other (modeling) languages the easiest, if there is a similar abstraction unit as a Simulink Subsystem, *e.g.*, subnets in Petri Nets [370], EClasses in Eclipse Modeling Framework [86], or methods/functions in textual languages. In fact, our approach may be more suitable in synthesizing valid instances in other languages, when one can construct clean interfaces without Simulink's 'spill-over' effects (see **RQ2.3**).

To illustrate the parallel in textual languages, consider the following example: depending on equivalence definition, the Java-methods `int min(int a, int b)` and `int max(int a, int b)` may be typed equivalent; similarly Python's `min(a, b)` and `max(a, b)` may be untyped equivalent.

Scaling and Accelerating Our Approach

Our strategies Random, Role, Bushy, and Deep use the algorithm of Listing 4.1 in their blueprint phase. It substitutes each Subsystem of the synthesized model once. Thus, in order to scale linearly, the functions `sample`, `isSuitable`, and the backtracking mechanism must produce constant overhead per Subsystem, as in Bushy, see Figure 4.8. For the models of our experiments, the construction phase scales effectively linearly, but is much slower for Random, Role, Bushy, and Deep. This is mainly because of Simulink's API for copying Subsystems or Blocks. In our experiment, the largest model's blueprint only took 30 seconds, but its Simulink construction 4 hours. As an additional scaling test, we also synthesized a blueprint-only model encompassing 2.4M Subsystems and 24M Blocks in 15 minutes using Bushy – its Simulink construction would take about 5 days. While Giant's blueprint phase is slower, its construction is much faster (see Figure 4.7) as fewer substitutions need to be performed.

We identified two factors to speed up the construction phase: (1) add each Subsystem's interface mapping to its properties in Section 4.1.4, as currently each substitution computes two interface mappings for the order relaxation: one for the Subsystem before substitution; one for the Subsystem after. (2) The construction (and all other phases) could be parallelized. Currently, our implementation is fully sequential, but the model synthesis steps are completely independent.

Threats to Validity

To ensure robust evaluation, we excluded the unstable models from the SLNET dataset (see Section 4.1.5) as they caused bugs in Simulink. If we use the whole, 'untamed' dataset, the large strategies Bushy, Deep, and Giant would mostly fail to produce any valid models. Excluding these models thus partially shifted our fuzzing from the synthesis phase to the earlier 'taming' phase (see Table 4.4), while simultaneously ensuring we can synthesize large models.

Our scalability results of Table 4.5 showed good fits ($R^2 > 0.6$) for four regression curves when applied to the 'tamed' dataset. However, 'taming' removed model elements (*e.g.*, Callbacks) that impact time/space complexity. We expect the original SLNET models to: (i) Run slower (due to Callback computations), (ii) Impede prediction based solely on element count (again, due to Callbacks), and (iii) Yield larger files (from restored elements) or smaller files (if library Blocks dominate). Thus, our regression may only reflect scaled-up behavior of the 'tamed' dataset – not the original models.

We treated some Subsystems with complicated or 'clunky' [314, 4] interfaces (Subsystems with Bus Ports, physical Ports, and Stateflow Charts) as common Blocks that are not substitutable. Including them would result in more equivalence classes, potentially as singletons.

4.1.8 Related Work

Fuzzing in CPS and other Domains

Methods for creating fuzzing candidates can be roughly divided into two categories: generative approaches, and transformative approaches [349]. In generative approaches, rules, or LLMs are leveraged to generate candidates from scratch, while the latter ones mutate existing seeds into candidates. A number of fuzzing techniques – mostly promising approaches from textual languages – have already been employed in the CPS domain [212, 235, 308, 326, 330]. Chowdhury *et al.* [212] developed CyFuzz (generates models up to size of ∼35 Blocks), a generative approach, which was inspired by CSmith [105] to fuzz Simulink's code generator. Shrestha *et al.* [308] then employ the long short term memory [36] approaches DeepSmith, and Deepfuzz [237, 269] (40 Blocks; 8 connected), and later improve it with a trained version of the LLM GPT-2 [278] to generate their models [326] (30 Blocks; 12 connected). With the rise of LLMs in modern textual language fuzzers [367, 368], we anticipate their adoption in the CPS domain in the future.

Prior transformative approaches in the CPS domain rely on equivalence modulo inputs (EMI) [152], which mutates unexecuted code in already compilable models and detects generated code changes. Note that in their context, *equivalence* refers to preserving semantics after mutation, whereas our *equivalence* ensures that two Subsystems can be substituted without invalidating the model. Using EMI, Chowdhury *et al.* [235, 291] developed SLforge, and later SLEMI (~3,000 Blocks). Both, Guo *et al.*'s COMBAT[330], and Li *et al.*'s RECORD[364], combine generation and transformation to first seed with SLforge, and then mutate with EMI (~3,000 Blocks).

The most comparable approaches to our work were not yet used in the CPS domain. Both Holler *et al.* and Han *et al.* [108, 264] mine building blocks from a program collection and then recombine them. Han *et al.*'s fuzzer, CodeAlchemist, recombines building blocks by putting fitting ones in sequence, the blocks can then grow in size to be recombined again (8 JavaScript statements). While CodeAlchemist does check block types, it can only put them in sequence. Instead, we put building blocks into 'suitable holes' of building blocks: in effect, we leave large local sub-ASTs intact. Already in 2011, Holler *et al.* proposed LangFuzz, which first mines AST-fragments to then replace a fragment in a program with another of the same type (no size given). In our work, we also replace a type of AST-fragments – Subsystems – but guarantee validity by only substituting equivalents.

Wang *et al.* [231] developed another related idea to construct more natural fuzzing candidates. Their fuzzer, SkyFire, learns the patterns from a code dataset, which are recreated in a probabilistic context-sensitive grammar (< 200 elements). Finally, Hussain *et al.* [191] presented RUGRAT, which generates Java programs from stochastic parse trees (< 5*M* lines of code, without linear scaling).

Model Synthesis

Models are also synthesized for scalability or stress tests. Semeráth *et al.* [325] synthesize valid and diverse Yakindu statecharts and EMF models by iteratively extending partial models until acceptance criteria are met. This is similar to our ground-up synthesis, though their approach is less scalable (300-1,000 model elements per hour). Nassar *et al.* [300] generated large EMF models, leveraging given metamodels for rule-based model transformations, which they iteratively apply. While their approach does scale super-linearly, it still can generate up to 500k model elements. He *et al.* [187, 188, 266] generate various model types in quadratic time, relying on constraint solvers and metamodels. In contrast, our approach does scale linearly and works without a known metamodel.

Clone Detection

There are various approaches to detect clones in models. Clone detectors may also cluster Subsystems into *clone classes* [126, 362]. These are only loosely related to our *equivalence classes* though: while Subsystems of the same clone class are similar in structure, substituting two Subsystems of a class would often invalidate the model. However, our equivalence classes guarantee substitutions work. Furthermore, most clone detectors are static and ignore dynamic aspects of typing or dimensions, which is central to our concept of typed equivalence.

4.1.9 Conclusion

Based on our concept of equivalence, we introduced Grandslam with five synthesis strategies – each generating models with Subsystem trees of different shape or size. All generated models are statically valid, and many are also compilable and simulable. Our approach uncovered eleven confirmed bugs and two confirmed scalability issues in Simulink, a safety-critical and mature tool. GRANDSLAM's linear scaling enables the generation of very large Simulink models, making it ideal for scalability tests.

In future work, we will extend GRANDSLAM to support more fine-grained substitutions. As discussed, our approach can be generalized to any group of Blocks. It could also be combined with another fine-grained mutator such as the modeling assistant SimGESTION from Adhikari *et al.* [362], which can mutate models one element at a time, or an LLM model completion tool like RAMC [365]. GRANDSLAM could first generate the model's rough outline (*i.e.*, the Subsystem tree) and then delegate the finer details to another approach.

Data Availability Our code with instructions is available at `https://anonymous.4open.science/r/grandslam`. Our data artifact package is uploaded to Zenodo with the reserved DOI `10.5281/zenodo.17296885`. Due to double-blind review constraints, we will publish it upon acceptance (see our provenance annotations in the synthesized models in Figure 4.4). The fingerprint of our uploaded package is `md5:c4ef50f1eb4f1d1a84e7805c5b57bc35`. The package's existence and integrity can be verified by the program chair or artifact evaluation committee upon request.

↪ Thesis Roadmap

4.2 SMOKE2.0 Whitebox Anonymizing Intellectual Property in Models While Preserving Structure

Authors: Alexander Boll, Manuel Ohrndorf, and Timo Kehrer.
Submitted to: Software and Systems Modeling.

Extends: *SMOKE: Simulink Model Obfuscator Keeping Structure*, Alexander Boll, Timo Kehrer and Michael Goedicke, In: *MODELS Companion '24: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, 2024.
Abstract: Simulink is widely used across various industries to model and simulate cyber-physical systems. Most industry-built models contain sensitive intellectual property, which prevents companies from sharing models with interested third parties, such as researchers or collaborating companies. However, advancing model-based engineering research requires access to such models – either to derive empirical insights or to evaluate new tools. While initiatives to replace industry-built models with open-source alternatives exist, they offer only a limited remedy. In this work, we introduce a novel approach with SMOKE, a Simulink obfuscation tool designed to selectively protect intellectual property within models. This allows companies to share relevant parts of their models with researchers or other third parties while safeguarding all sensitive information. SMOKE's whitebox design preserves the model's original format and structure, ensuring that meaningful insights remain accessible. We evaluated the tool on an extensive set of open-source models and found it successfully removes sensitive components, while preserving model structure. A video demonstration of SMOKE is available online at youtu.be/0i42BzgJAUA.

4.2.1 Introduction

Across various industries, Simulink is a widely-used tool to design, implement and simulate cyber-physical systems [208, 1]. The popularity of Simulink also gives rise to a considerable research interest, *e.g.*, into better understanding Simulink models and their evolution [10]. However, within the research community, it is well known that companies often refrain from sharing their Simulink models to protect intellectual property (IP) [2]. In some cases, industry partners share their models under severe limitations, such as non-disclosure agreements, which strictly prohibit the publication of model files or any visual representations of the models. Companies may also restrict access to models to company computers and locations. This practice creates significant challenges for the FAIR[2] principles [201] in Simulink research and often leaves researchers without useful study subjects [10]. The modeling research community has begun addressing this issue by developing corpora of open-source Simulink models [1, 337, 10, 5], which can serve as substitutes for proprietary models in research. In general, however, open-source models are much smaller than industry models, and are often no adequate substitutes [1, 2].

In this work, we introduce SMOKE: **S**imulink **M**odel **O**bfuscator **K**eeping structur**E**, an extensible, open-source tool for protecting intellectual property through selective anonymization of Simulink models. Our goal is to anonymize models by removing sensitive information while preserving their general usefulness, especially for research. Thus, a model being anonymized using SMOKE is still a valid Simulink model whose logical structure is isomorphic to the original one, yet exposing a different visual appearance and functional behavior, depending on the desired degree of anonymization. Both kinds of modification are implemented through unidirectional model transformations [59], *i.e.*, they are non-reversible without logging or domain knowledge. The first class of modifications are layout obfuscations, which preserve functionality but reduce understandability. The second class comprises sanitization techniques, which remove sensitive data or functionality. This is a concept we adapt from database sanitization, originally developed for relational databases [51]. The concrete transformations to be applied can be selected by the user, depending on the desired degree of anonymization. SMOKE supports both interactive and non-interactive selections of transformations, which are then composed to an executable transformation workflow.

SMOKE addresses two use cases which are of primary interest for researchers. First, it simplifies obtaining approval for publishing visual representations of Simulink models by selectively removing layout information. A precursor to SMOKE was already previously used to obfuscate industry models, enabling the publication of screenshots in scientific articles [247, 293]. Second, companies may permit further study or use of sanitized models where sensitive data or functionality is removed. As opposed to traditional obfuscators concealing a program's [35] or model's [359] entire functionality within an uninspectable and immutable virtual black box [44], our white-box anonymization yields a native Simulink model open to further inspection. This means that the model can be opened with the standard MATLAB/Simulink editor or other tools working with Simulink models. A particular feature of SMOKE is that it preserves the logical structure of the original models. Even if highly anonymized, the obtained 'structure-only' models still remain valuable study

[2] Findability, Accessibility, Interoperability, and Reuse of digital assets.

subjects for many research interests. To better understand various aspects of a model or its evolution, empirical research on Simulink models often focuses on metrics related to model structure [236, 1, 337, 4], or relies on third-party analysis tools such as clone detectors, differencing tools, slicers, or variant and information flow analyzers that require an intact model structure to function [116, 177, 230, 251, 273].

In addition to addressing researchers' interest, SMOKE can also be employed by industry partners themselves. In software value chains, models are developed in co-engineering fashion, and such collaborations only work if the parties have access to the models [357]. However, companies may still hesitate to grant full access to others or may need to comply with various competition laws [219]. Moreover, (partially) anonymized models can be indexed by model search engines, as they can search models by basic metric structure [303, 334, 5]. This way, interested parties may find models according to basic information, and can get into contact with the owners to agree on the terms of full access.

We give an overview of SMOKE's anonymization capabilities, and showcase the effect of interactively applying a subset of those on a realistic example. Moreover, we report about our evaluation applying SMOKE's full anonymization capabilities on thousands of open-source models taken from SLNET [337], with a particular focus on evaluating its general applicability and functional correctness. While SMOKE focuses on Simulink models, we argue that other modeling ecosystems, such as UML, SysML, Modelica, etc., could also benefit from a layout or functionality anonymizer [299].

Our tool SMOKE, written in MATLAB, along with its documentation, and all artifacts from our experimental evaluation, are open-source and available at github.com/lanpirot/SMOKE. A brief video demonstration of SMOKE is available online at youtu.be/0i42BzgJAUA.

This paper is a substantial extension of the short paper "SMOKE: Simulink Model Obfuscator Keeping Structure" by Boll *et al.* [11], demonstrated at the MODELS'24 demo track. We have built upon our prior work through the following improvements:

- We updated SMOKE, extending it with new anonymization functionality. Notably, it is now applicable to library models, and anonymizations can be performed on user-selected scopes of the model.
- We repeated our entire evaluation, incorporating the aforementioned new functionality and now including library models. We greatly improved the evaluation of the behavior (non-)preservation of SMOKE.
- We further added discussions on SMOKE's coverage and threats to validity.
- Leveraging the expanded space available compared to the original short paper, we have meticulously rewritten and significantly enhanced all sections, notably Sections 4.2.2 to 4.2.5, from scratch. This revision includes extensive additional detail and thorough explanations that were not feasible to incorporate previously.

4.2.2 Background on Simulink and Obfuscation Techniques

In this section, we first lay the foundational background knowledge of Simulink that is needed for this paper. We then elaborate on obfuscation and sanitization techniques, and

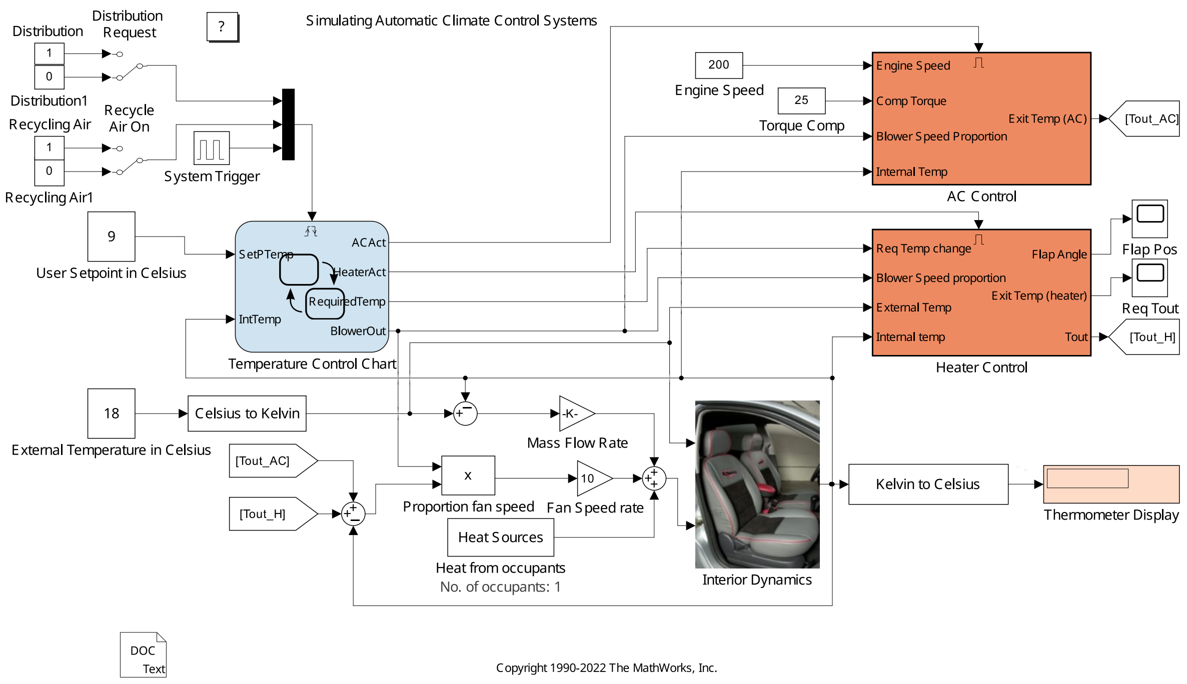


Figure 4.9. An exemplary Simulink model (with the name sldemo_auto_climate) showing various kinds of Blocks, connected by Lines.

how they can be applied in Simulink. Note that we do not discuss a completely exhaustive list of obfuscation techniques, but focus on those that are relevant to our context, *i.e.*, the ones that are applicable to Simulink models, while keeping structural integrity.

Simulink

Simulink [242] is a modeling language and versatile integrated development environment developed by Mathworks. It is based on and integrated into the MATLAB IDE and can be used for abstract modeling, implementation of functionality, simulation, and code generation. As its graphical modeling language is intuitive to use and understand, it is often used as a low-code development platform [328] in non-programmer domains like automotive, aerospace, medical and other technical industry domains [83, 114, 208].

A Simulink model consists of *Blocks*, which can be connected by *Lines* – both placed on a modeling canvas, see Figure 4.9.[3] Lines transport values from Block outport(s) to Block inport(s). Blocks transform their input from their inport(s) to an output which they emit via their outport(s). As a model grows, it can be partitioned using special *Subsystem* Blocks that nest Blocks and Signals, allowing developers to hierarchically structure their models. Subsystem Blocks can also nest other Subsystem Blocks and with this, developers can hierarchically structure their models.

The Simulink IDE offers different model *views* via the Subsystem Blocks. In a view, only the direct content of the currently selected Subsystem is visible, while Subsystems hide their

[3] Example Simulink model from mathworks.com/help/simulink/slref/simulating-automatic-climate-control-systems.html (visited on 15/12/2025)

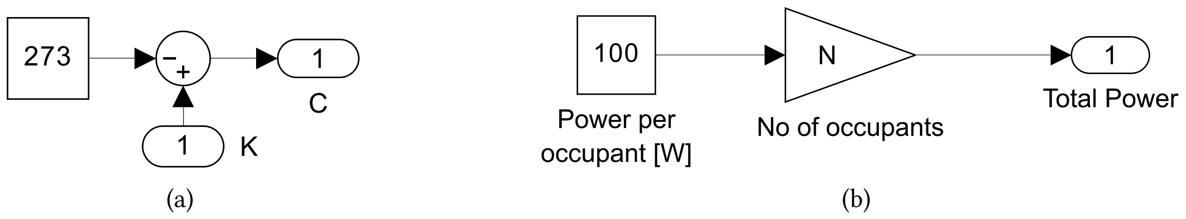


Figure 4.10. The inner view of two Simulink Subsystems of the model in Figure 4.9: the Subsystem in Figure 4.10a transforms temperature from Kelvin to Celsius: $C = K - 273$, the Subsystem in Figure 4.10b computes the total heat generation of N occupants: $TotalPower = 100 \cdot N$.

nested implementation details from the outer view. While Figure 4.9 depicts the root view of a model, the Subsystems' inner views of the Subsystems 'Kelvin to Celsius' (lower right of Figure 4.9) and 'Heat Sources' (bottom middle of Figure 4.9) is given in other views, shown in Figure 4.10. The root Subsystem, *i.e.*, the model itself, may also have inports and outports, which act as the input and output of the model. In many models, inports are driven by physical sensors and outputs drive actuators. Additionally, there are also models without input or output, *e.g.*, library models that contain useful sets of custom Blocks, or functionality used in other models where they can be imported and reused. While Simulink already offers an extensive set of different Block types, users can use this mechanism to define their own (complex) Block behaviors. In addition, each Simulink Block comes with its own set of *Parameters*[4] that can be adjusted individually for each instance of the Block, see Figure 4.15a. This variety of Block types and their individual Parameters offer a comprehensive range of design options supporting both static modeling (memory-less systems), and dynamic modeling (time-evolving systems with states).

In this work, we define the model's *structure* as a typed graph of Blocks connected by Lines (see Figure 4.11). When we speak of *structure-preserving* anonymizations, or transformations, we are thus looking for obfuscations and sanitizations that do not alter the underlying graph of Blocks and Lines.

Obfuscation Techniques

Obfuscations are transformations that change aspects of a program, while preserving its functional behavior. Obfuscating a program – in our case 'a program' is to be understood as 'a model' – increases the difficulty of understanding, or accessing its purpose or logic [145]. Using obfuscations, one can thus protect the intellectual property of a program and make it harder to reuse or repurpose it.

Collberg *et al.* [35] identify three basic classes of obfuscation: (1) *layout obfuscation*, (2) *data obfuscation*, and (3) *control obfuscation.* (1) Layout obfuscations are simple transformations that adapt documentation, names, or formatting, usually by removing them, resetting them to default values, or replacing them by arbitrary values. As Simulink is a graphical language, it offers many more layout obfuscations than classical textual languages, *e.g.*, Blocks and Lines can be repositioned, resized, or recolored, fonts can be changed, and media

[4] In EMF models, the counterpart of Parameters are called Attributes [282]. The Parameters of a Block define how it exactly behaves during simulation, and are not external output/input like parameters in textual programming languages.

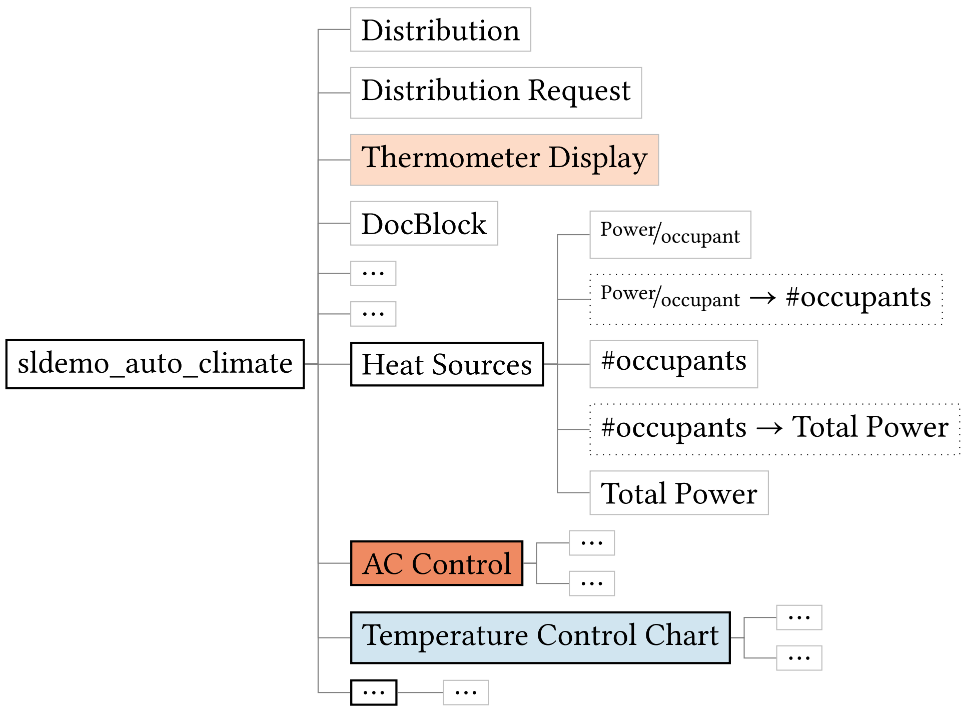


Figure 4.11. The partial structure of the model of Figure 4.9. All structure elements are part of this tree, hierarchically organized by the Subsystems. The root Subsystem is on the very left. Each Subsystem's child is a Block or Line within it. Only the content of the Subsystem 'Heat Sources' (see Figure 4.10b) is fully shown, all other content is only hinted at.

elements can be used. (2) Data obfuscations alter the storage, encoding, or ordering of data. A different encoding, or even encryption, makes it difficult to interpret the data that is being processed in the program. Additional data abstraction transformations can be splitting up related data, or bunching up data that is unrelated. Simulink offers multiple data types, like `int8`, `double`, `boolean`,[5] with which one could obfuscate data encoding. Additionally, transformations on data abstraction are possible by using BUS-Blocks, that bundle up data from multiple Lines into one. (3) Control obfuscation manipulates the abstraction layers, adds useless conditions, or dead code. Obfuscating the abstraction layers is similar to the obfuscation of data abstraction: the removal of meaningful hierarchy layers or the addition of extraneous ones. In Simulink this can be achieved, *e.g.*, by resolving Subsystems or adding new arbitrary ones into the model.

Sanitization Techniques

While obfuscation does not change the behavior of a program, *sanitization* does. In this work, we generalize the concept of *data sanitization* [39, 51] to our purpose of *model sanitization*: data sanitization is applied on sensitive data in relational databases by selectively removing parts of the database, while valuable insights into the rest of the data still remain possible. Data sanitization is performed so that the protected database can be shared with others, without risking the sensitive data being leaked.

[5] It is further possible to create new, custom data types.

In our case, we want to preserve the structure of a model, so that valuable insights into the model are preserved, while the model behavior may be removed, changed or even completely broken. This means, the model should still be syntactically correct after all sanitization transformations, and still be loadable and inspectable in the IDE. On the other hand, the sanitized model should not show the same behavior, which means it either does not compile anymore, the simulation crashes or stops prematurely, or, given the same input, the output of the model changes.

As the structure of the model shall remain stable, the only way to alter the model functionality is to change Block or model Parameters. Such changes either result in altered behavior of Block calculations and thus overall model behavior, or model misconfigurations that lead to non-compilability or simulation crashes.

4.2.3 Related Work

Simulink-specific Obfuscation

While obfuscation in traditional text-based programming languages is a well-established discipline with decades of research [35, 145], we found only limited prior work on the obfuscation of Simulink models.

Simulink itself features the Protected Model Creator,[6] a versatile tool that transforms Simulink models into black boxes of another file format, or white boxes that are not editable. However, these immutable white boxes do not anonymize at all. Other built-in options to create black box versions are so-called S-Functions or static libraries.[7]

A subset of SMOKE's layout obfuscations could be found in a tool by Ohashi[8] and the Obfuscate-Model tool by Jaskolka *et al.*[9] The former, however, is unmaintained and broken since at least 2017, according to the tool's reviews on MATLAB FileExchange. The latter is more mature and has been utilized for obfuscation in an industry-research partnership [247, 293]. Though it supports only layout obfuscations, we built SMOKE based on a fork of the Obfuscate-Model tool, extending it with new capabilities, refined prior capabilities, and fixed bugs. For a detailed list of all changes and improvements, see Section 4.2.4. Moreover, we evaluated SMOKE's functionality on a large model set.

Obfuscating Other Model Types

Obfuscation approaches are also suggested in various other domains, such as conceptual models, business process models, or ML models. An overview is given by Tevajärvi [359] who recently conducted a literature review of existing model obfuscation techniques and then compared them in terms of a case study. Among others, they successfully used functional mock-up interfaces and obfuscating generated code to preserve functionality while hiding sensitive data or functionality. However, the resulting black boxes are then in a different, opaque format.

[6] `https://www.mathworks.com/help/rtw/ref/protectedmodelcreator.html`

[7] `https://www.mathworks.com/matlabcentral/answers/91537-how-do-i-protect-the-ip-of-my-simulink-model-when-sharing-it-with-others-who-may-include-it-in-thei`

[8] `https://www.mathworks.com/matlabcentral/fileexchange/54359-model-obfuscation-tool`

[9] `https://github.com/McSCert/Obfuscate-Model`

Fill [107] discusses obfuscating conceptual models with the intent of making them shareable, but his transformation breaks the structure by splitting models into multiple parts. Other transformations he suggested break the model's syntax. Nacer *et al.* [195] also explore business process model obfuscation by fragmenting models.

A further concept of model sharing in a co-engineering scenario with multiple roles (see Martinez *et al.* [270]) is explored in the work of Weber *et al.* [366]. In a first step, the model is split into sub-modules and their interfaces. Each sub-module is encrypted and only decryptable by authorized parties – effectively fragmenting the model. In our tool, selective aspects of the model (*e.g.*, whole sub-modules or only sensitive properties within the model or some sub-modules) are removed.

Sihler *et al.* [357] build an obfuscation tool for IRIS, a graphical modeling tool for technology road maps (TRM). The tool can perform various one-way transformations with the intent of preserving model behavior, in addition to the model still being in the same file format. During their transformations, the model structure becomes completely removed, though. Their transformations provably uphold criteria, such as self-containment of the model parts that are left; no inference possible on the obfuscated model parts; and preserving behavior. This makes their tool similar to SMOKE's use case of obfuscation, while preserving behavior. However, SMOKE also serves the use case of selectively sanitizing behavior and preserving model structure.

Less related are the ideas of Gupta *et al.* [216], who protects CAD models by inserting sabotaging elements into the model that hamper reproducing physical instances of the model. One may interpret this as the model structure largely being preserved, while its behavior is changed.

Finally, the *ModelObfuscator* by Zhou *et al.* [361] is intended for Deep Learning models and uses various obfuscation techniques, some of them structure-preserving. However, many of them intentionally alter the model's structure, such as extra layer injection.

4.2.4 Approach and Tool

Approach

The main goal of SMOKE is to support users with the selective removal of sensitive intellectual property from their models via obfuscation, or sanitization, while keeping the model structurally intact and loadable. We thus create one-way and resilient [35] (aka. unidirectional [59]) obfuscation and sanitization transformations in Simulink that fulfill these criteria for SMOKE.

To come up with a list of possible transformations, we first went over the metamodel approximation of Simulink from the Massif group (referenced in [282]) and identified model elements, that could be altered to obfuscate or sanitize. In a second step, we consulted the Simulink Documentation for model elements that are not listed in this metamodel (*e.g.*, 'Model Information'). In a last step (see Section 4.2.5), we inspected the raw model files for possibly sensitive user information in elements, that were still present in models after we applied transformations.

As Simulink is a vast ecosystem,[10] we can't offer an obfuscation or sanitization option

[10] In addition to the hundreds of blocks Simulink already ships with, various toolboxes offer additional

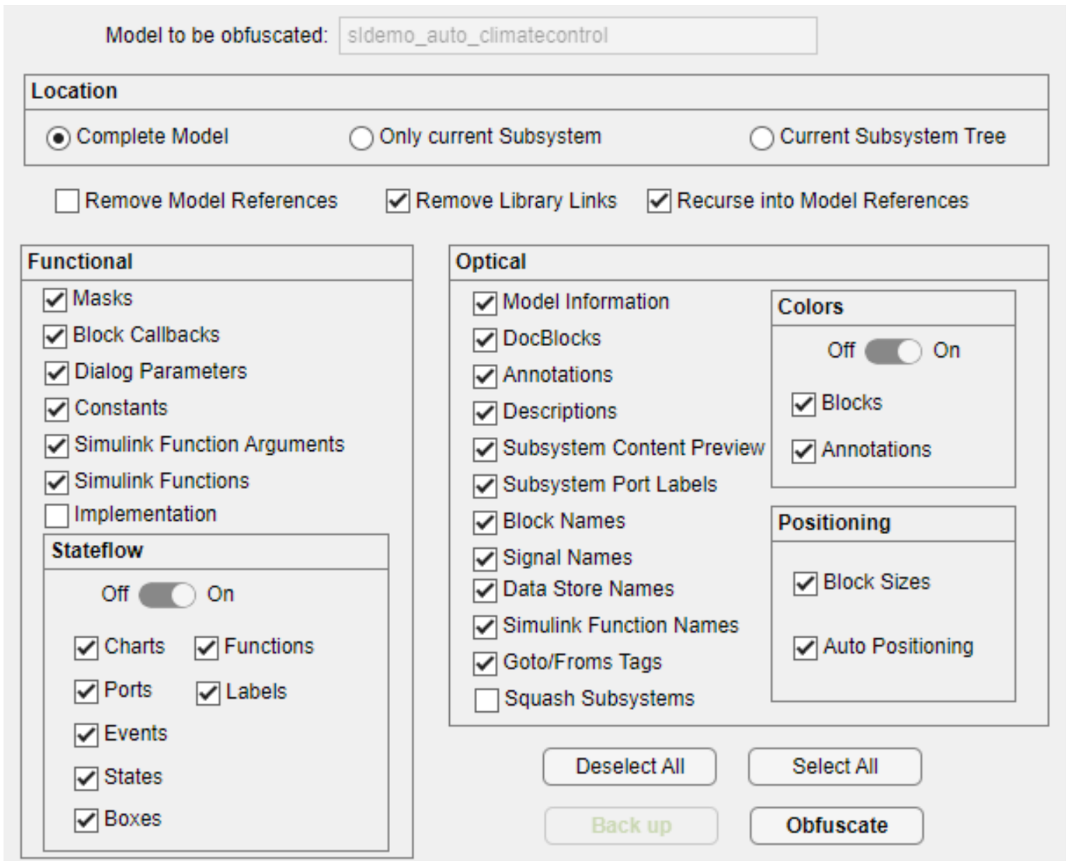
for every model element. Many model elements interact with each other in complex ways, which is impossible to deal with in a research prototype. Instead, we chose to offer options for those transformations that we deem the most intuitive ones: transformations on elements which are used often, are prominently displayed to users, or are used to hold sensitive or model-wide information. In Section 4.2.4 we will see though, that SMOKE can be easily extended to obfuscate or sanitize other model elements as users may seem fit.

Our extensive list of selectable elements to be obfuscated and/or sanitized can be seen in the main menu of SMOKE in Figure 4.12: in the lower right block 'Optical' is the list of obfuscations, in the lower left block 'Functional' is the list of sanitizations. In the following, we will shortly elaborate on a selection of these shown transformations, and our rationale behind them.

Obfuscations In Simulink, a model's primary Parameters, such as `CreatorName`, `CreationDate`, and `ModifiedBy`, are stored in its 'Model Information' Parameters. SMOKE can remove these Parameters, *i.e.*, reset them to default values, and thus disrupt the model's traceability. Next, Simulink offers various ways of commenting and documenting within models, such as 'DocBlocks,' 'Annotations,' and 'Descriptions'. Note that these documentation options are directly embedded into the model file, while additional model 'Notes' exist [6]. By removing such documentation SMOKE can impair the model's understandability. Additionally, Simulink offers a feature called 'Subsystem Content Preview,' which allows users to preview the contents of a Subsystem, such as the aqua Subsystem shown on the left of Figure 4.9. Disabling this preview ensures that no external information is visible in the current model view, meaning screenshots will only display information from the currently opened Subsystem. Furthermore, all other obfuscation options shown in Figure 4.12 either remove custom names or custom design elements from the model, both of which negatively impact the model's understandability.

Sanitizations In Simulink 'Masks' are used to hide and protect the internals of a Subsystem. This can be a simple picture icon visually (*e.g.*, the car seat picture on the Subsystem in the lower bottom of Figure 4.9), but can also offer complex custom Parameters. SMOKE removes such Masks, and the bare Subsystems remain under them. 'Block Callbacks' are custom MATLAB scripts that are executed on certain events, like loading the model or changing a view. They are powerful and can, *e.g.*, check for the presence of certain model elements, and depending on the result of the check, prevent the saving of the model. Apart from a few undeletable Callbacks, SMOKE can remove all others. 'Dialog Parameters' are probably the most important sanitizer, as all kinds of Block functionality can be changed here for each Block individually (see Figure 4.15). SMOKE will reset Parameters to Block default values, whenever possible. 'Constants' are special Blocks, that drive a constant input into the model. Constant Blocks' values are rest to either default number or string values, depending on their type. More interestingly, users can embed a whole MATLAB script into a Function

Blocks. Each Block type has its own specific Parameters in addition to the common ones. We observed more than 100 Block types and more than 10,000 Parameters. Finally, Simulink comes with its own sub-languages: StateFlow, and SimScape. This results in a multitude of corner cases, that need to be dealt with, if one strives for an exhaustive obfuscator.



Model to be obfuscated: sldemo_auto_climatecontrol

Location

[✓ Complete Model] Only current Subsystem Current Subsystem Tree

[ ] Remove Model References [✓] Remove Library Links [✓] Recurse into Model References

Functional

- [✓] Masks
- [✓] Block Callbacks
- [✓] Dialog Parameters
- [✓] Constants
- [✓] Simulink Function Arguments
- [✓] Simulink Functions
- [ ] Implementation

Stateflow

Off [✓ On]

- [✓] Charts
- [✓] Functions
- [✓] Ports
- [✓] Labels
- [✓] Events
- [✓] States
- [✓] Boxes

Optical

- [✓] Model Information
- [✓] DocBlocks
- [✓] Annotations
- [✓] Descriptions
- [✓] Subsystem Content Preview
- [✓] Subsystem Port Labels
- [✓] Block Names
- [✓] Signal Names
- [✓] Data Store Names
- [✓] Simulink Function Names
- [✓] Goto/Froms Tags
- [ ] Squash Subsystems

Colors

Off [✓ On]

- [✓] Blocks
- [✓] Annotations

Positioning

- [✓] Block Sizes
- [✓] Auto Positioning

Figure 4.12. The main menu of SMOKE. The menu's elements are described in Section 4.2.4.

Block, which are sanitized, *i.e.*, their body is removed by activating 'Simulink Functions'. We also address various parts of the Simulink sub-language 'Stateflow' which offers to embed state machines or stateflows into Blocks, with more fine-grained options.

Structural Transformations The options 'Squash Subsystems' and 'Implementation' (the only unchecked options in the bottom of the panels 'Functional' and 'Optical' in Figure 4.12) do in fact change the model structure. We still incorporated them into SMOKE, as they were recommended by users. 'Squash Subsystems' removes all Subsystems, *i.e.*, they get resolved and a completely flattened model hierarchy emerges. When checking 'Implementation' all Blocks and Lines apart from Inports and Outports of the current Subsystem get removed. This option is useful when certain parts of the model can be shared, while other parts are so sensitive that even their structure needs to be removed.

Reversing Transformations In general, obfuscation and sanitization transformations should be hard or impossible to reverse; otherwise, they need not to be applied. We thus implement (see Section 4.2.4) all our transformations to actually remove or reset, and not to just hide model information. In particular, we remove model elements, whenever possible. Elements, whose removal would result in a changed structure, or broken model, are reset to default values. As such custom information is removed, it can only be reconstructed using additional domain knowledge or by extrapolating other model information that was not selected to be removed by the user.

Additionally, we made sure, that model information is actually permanently removed, once a transformation was applied by the user. There appear to be only two possibilities for reversing any deletion: either the deleted information, while not visible to the user in the Simulink IDE, may still be part of the raw model file (1), or the Simulink IDE may offer an "undo action" functionality to restore the information (2).
Regarding (1): we compared pairs of models before and after transformations, manually and using scripts, to make sure the deleted information is indeed removed from the raw model file. We found our transformations to completely remove the selected information from the raw file. As the model information is saved into and recreated from this file, this path of transformation reversal is thus impossible.
Regarding (2): Simulink does offer to undo IDE actions from its current session. Once a model is saved and closed in the IDE, the edit history dissipates and is not recoverable from the saved file.[11] Furthermore, the transformations SMOKE calls are not reversible in the first place, as they are not IDE edits, but edits executed by scripted MATLAB transformations, as shown in Section 4.2.4. MATLAB can currently only undo IDE edits, though.
In conclusion: we are positive, that sharing a SMOKE-transformed file will remove all the user-selected information permanently and irreversibly.

SMOKEing other Modeling Languages Most ideas we present in this paper, are similarly applicable to obfuscate or sanitize models created in other modeling languages. One simply has to choose what kind of information shall be preserved in the model (in our case, the model structure and ability to parse it with the IDE), and what types of information can or should be removed. For modeling languages with a known metamodel and fewer Block types and Parametrical corner cases than Simulink, it may also be possible to automatically derive obfuscation/sanitization transformations directly from the metamodel. Mathworks, however, never published Simulink's metamodel, and only the unofficial approximation by the Massif group [282] exists.

Implementation and Extensibility

We implemented SMOKE[12] in the MATLAB scripting language as a standalone application. Every transformation uses the Simulink model API of MATLAB, to alter the model via functions such as `delete(<modelElement>)`, or `set_parameter(<blockID>,<Parameter>, <newValue>)`. We did not directly alter the model's raw XML files, as this risks breaking the model, *i.e.*, making it impossible to load or edit the model.

After starting the SMOKE app, the user chooses a Simulink model as input, and then selects transformations (see Section 4.2.4) to be applied on the model. SMOKE transforms a model through a pipeline architecture, see Figure 4.13. If the user chooses multiple transformations, they will be applied sequentially in a pipeline architecture. Before any transformation takes place, the model and all its Blocks are unlocked so they are editable in the further process.

[11] This applies to MATLAB versions through R2025a and is expected to remain valid in future versions.
[12] Available at `https://github.com/lanpirot/SMOKE`.

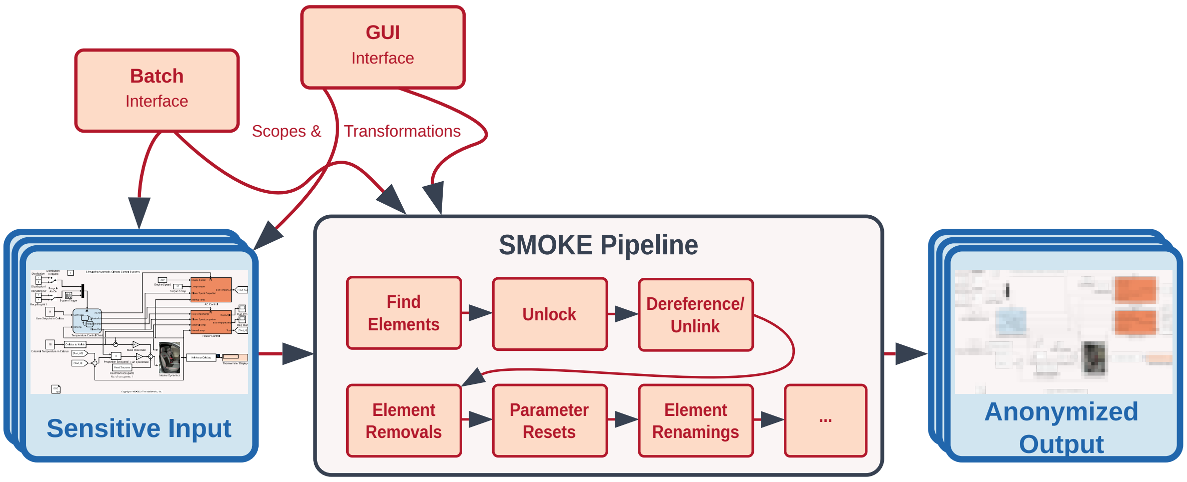


Figure 4.13. Smoke's architecture: its pipeline of model transformations can be invoked from the GUI or in batch mode.

Next, the model references are resolved and library links are removed, as edits to them are not allowed by Simulink, otherwise. Finally, all selected transformations are applied sequentially.

Each transformation iterates over all the elements it pertains to and transforms them one by one, *e.g.*, the DocBlock removal iterates over all DocBlocks. To ensure that Smoke preserves the model structure, every transformation that is called removes exactly one model element or resets exactly one Parameter at a time. We only ever remove Annotations and DocBlocks (they are not structurally relevant) and check whether a Parameter change affects the model structure before attempting to change it. We thus guarantee structure-preservation by construction. For most transformations, our implementation is trivial, with just a few necessary interventions to keep the model intact, *e.g.*, names of Blocks have to be unique in each view, or references to names have to be updated appropriately. For the resetting of names, we choose a trivial naming scheme, *e.g.*, every Block will get the name `<block type of Block><Number>`.

One of the more interesting transformations is `removeDialogParameters.m`. Here, all Blocks' Parameters with custom values shall be reset. Simulink does not offer a direct functionality for such a reset, and also does not give default values to which to reset them to. We thus create and insert a dummy Block of the same type into the model, and read out, and copy its values for Parameters. Sometimes dummy Blocks do not have all Dialog Parameters of the actual Block (c.f. also Figure 4.15, where the Parameter list changes after transformation). In this case, we reset Parameters that are numeric to `0` and string type to the empty word `''`. Further, we preserve some Parameters of Blocks, that would alter the model structure: we found, *e.g.*, 7 Parameters that change the number of ports of a Block. Resetting them could remove the incoming or outgoing Signal Lines of the Block and thus break the model structure. After the Block's Parameters are reset, we clean up the model by removing the temporary dummy Block again. To reset a Block's size, we use the same technique and copy a dummy Block's default dimensions.

Our transformations' implementation is *optimistic* [33], *i.e.*, our transformations may try to change or remove model elements that could break the model. Breaking transformations are caught by Simulink's error detection though and we simply skip the transformation for elements causing such errors, thus leaving the model syntactically correct. Some examples that get caught are: certain Blocks are not resizable; some model elements cannot be removed as Callback Functions are dependent on them, etc.

As each transformation takes and leaves a structurally intact model, most transformation orderings are interchangeable. For best performance, SMOKE still performs according to a partial ordering: It is best to start with transformations that remove model elements, like Blocks. This ensures that no transformation is performed on elements that are later removed, anyway. The removal of sizes and positions should be performed last. Resetting the shapes and sizes of Blocks, in the penultimate step, ensures that potential text can still be completely displayed. The removal of positions should be the last transformation: here, all Blocks and their sizes are taken into account to achieve visually pleasing diagrams without any overlap of model elements.

For debugging purposes, we found it best to move renaming transformations to the back, as it is much easier to quickly match the original model and its transformed counterpart this way. For an easier comparison of the model pair, it's also best to deselect size and position removal, as then all elements 'stay in their place' and can be matched visually at a glance.

To extend SMOKE with new transformations, users can either: (1) add a transformation to their suitable transformation list within 'element removals', 'parameter resets', and 'element renamings' (the lower middle boxes in Figure 4.13); (2) append it in the '...' box. The sole requirement for any new transformation is that it must preserve syntactic correctness – that is, it should accept a valid Simulink model as input and produce a valid Simulink model as output, consistent with all prior transformations.

Comparison of SMOKE with Prior Versions Our application is based on a predecessor app called `Obfuscate Model`[13] by Jaskolka *et al.* Here, we first compare `Obfuscate Model` to SMOKE in the state of the publication of [11], *i.e.*, SMOKE1.0. Next, we compare SMOKE1.0 to SMOKE in this paper's state, *i.e.*, SMOKE2.0.

Jaskolka *et al.* built a Simulink model obfuscation tool written in MATLAB. Besides some bug fixes (*e.g.*, crashes on broken array indices), adding exception handling, and a slight redesign of the UI, in SMOKE1.0 we:

- added various functional transformations. `Obfuscate Model` was restricted to graphical obfuscations only.
- added the graphical obfuscations for resetting block position and block sizes.
- enabled deep obfuscations, where SMOKE1.0 also descends into masked, protected, variant or linked Blocks. These were not transformed before.
- added a "back up" option to quickly reset the transformation process and recover the original model.

[13] https://github.com/McSCert/Obfuscate-Model

Since [11], we fixed some further bugs and sped up SMOKE2.0's execution, as well as:

- The user can choose the location or scope where to apply transformations. This way, some parts of the model can be kept in their original state or be transformed differently than others.
- SMOKE2.0 can be applied to locked (library) models, which makes it applicable to all types of Simulink models, now.
- We added the most radical transformation: removing the complete implementation of a Subsystem. The whole local Subsystem tree is thus pruned.
- We added an obfuscation that removes the Subsystem hierarchy, and thus "flattens" the model, as all model elements are put into the same hierarchy level.
- We added a sanitization that removes Simulink Function bodies.

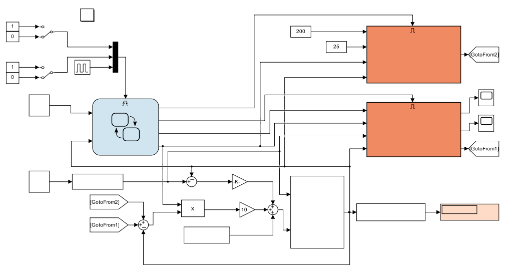
SMOKE in Action

Menu and User Interaction The menu screen of SMOKE is presented to the user right at startup. At startup, the boxes are pre-checked as shown in Figure 4.12. In the upper line, the model slated for transformation is shown. For this, SMOKE automatically chooses the last model that is opened by the user. All chosen transformations will later be performed on this model. Next, in the second line, the user can select the scope of all chosen transformations. Either the complete model gets transformed, only the currently selected Subsystem and its direct elements, or a whole Subsystem Tree, *i.e.*, the current Subsystem and its descendants. After that, users choose how to handle model references and library links. In the lower left box, all options that remove model functionality, *i.e.*, sanitizations, are listed. In the lower right box, all options pertaining to obfuscations are listed, which leave model functionality intact. On the bottom right are the action buttons. Two buttons for convenience that check all or none of the boxes. The 'Obfuscate' button triggers the obfuscations and sanitizations the user chose. The 'Back up' button reverts the model to its original, *i.e.*, last saved state.

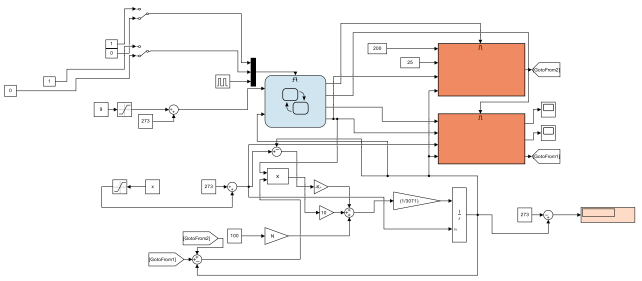
We envision two main user journeys using the GUI: (1) apply all pre-selected transformations as shown in Figure 4.12, which removes everything but the main model structure, (2) start with few transformations and selectively apply more and more of them, perhaps in different scopes, until all sensitive information is removed. When the user is satisfied with the model's state, they can save it, and it is ready to be shared.

In addition to SMOKE's interactive mode, it can also be called via MATLAB's scripting API. In this way, a whole project's models can be anonymized in batch mode, using pre-selected transformations. We used this mode in our evaluation in Section 4.2.5 to handle thousands of models at a time.

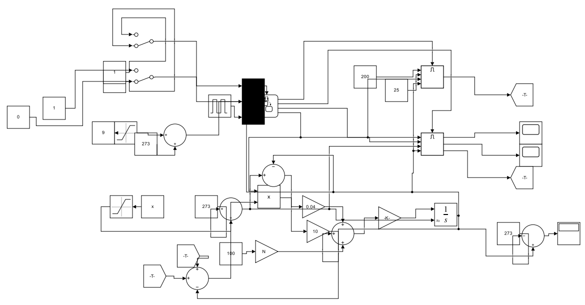
Exemplary Obfuscation To give an intuitive overview of the capabilities and impact of SMOKE, we give an exemplary user journey of a model's obfuscation (Figure 4.14) and sanitization (Figure 4.15) of the model from Figure 4.9. In a first step, a user chooses to remove all documentation elements, labels and names, which leaves a completely anonymous model in Figure 4.14a. The model's layout otherwise is completely unchanged, and a visual mapping



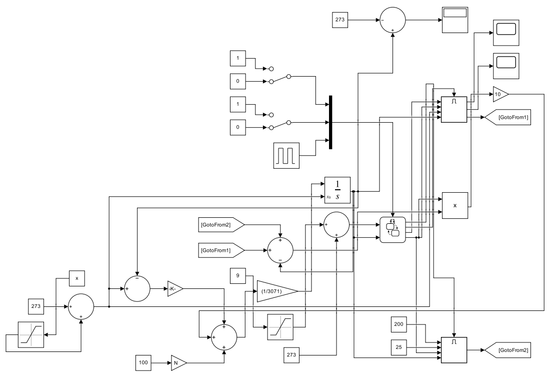
(a)



(b)



(c)

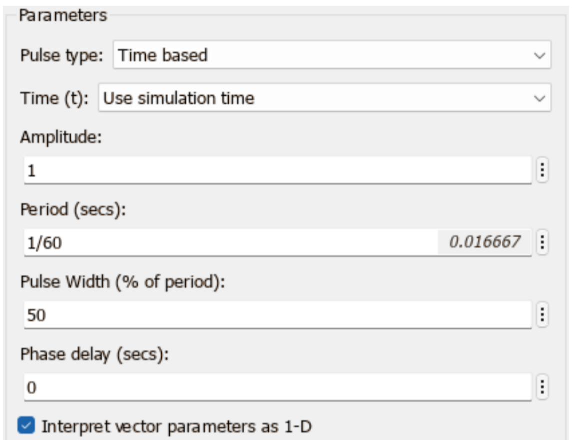


(d)

Figure 4.14. Step-wise obfuscation of the model from Figure 4.9. First, Annotations, Docblocks, labels and names are removed in Figure 4.14a. The Subsystem hierarchy is flattened in Figure 4.14b. Then, the Block colors and sizes are reset in Figure 4.14c. Finally, in Figure 4.14d the Block and Line positions are reset.

from the original is trivial. Next, the user decides to further flatten the model's hierarchy and all Subsystems in the model get resolved. This affects some Subsystems on the left and lower side of the model, which get replaced by their inner Blocks in Figure 4.14b. To further obfuscate the model, the user removes the Blocks' colors and resets their sizes in Figure 4.14c. In a final step, the user obtains a clean model layout by using Smoke's autopositioning feature, with which one can cycle through semi-random layout arrangements. The final version is now completely obfuscated, while preserving the model structure and functionality. The original model from Figure 4.9 is hardly recognizable in its final form in Figure 4.14c.

If the user decides to (perhaps additionally to the obfuscation) sanitize the model, they can choose to apply various transformations affecting functionality (see lower left options in Figure 4.12). Most of these effects are not immediately visible in the IDE view, in contrast to



Parameters

Pulse type: Time based

Time (t): Use simulation time

Amplitude:

1

Period (secs):

1/60 0.016667

Pulse Width (% of period):

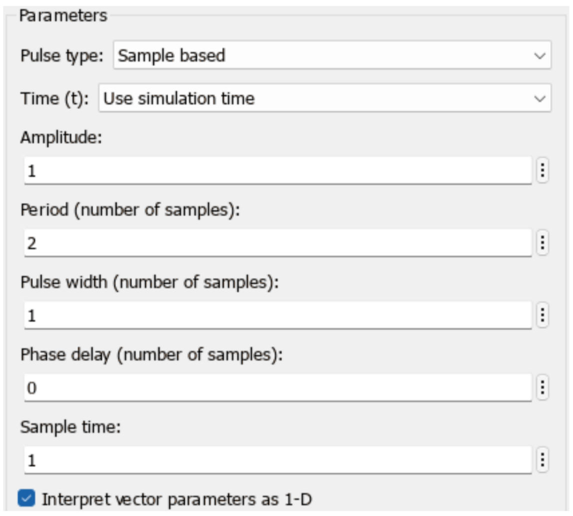
50

Phase delay (secs):

0

☑ Interpret vector parameters as 1-D

(a)



Parameters

Pulse type: Sample based

Time (t): Use simulation time

Amplitude:

1

Period (number of samples):

2

Pulse width (number of samples):

1

Phase delay (number of samples):

0

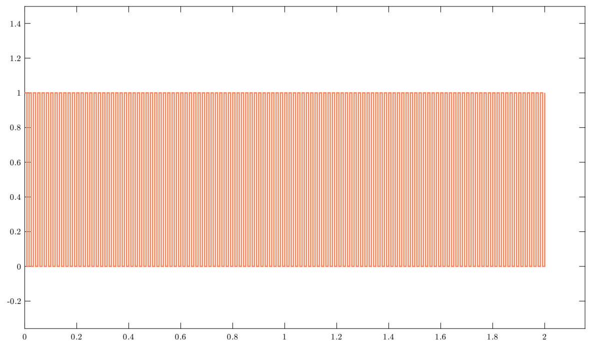
Sample time:

1

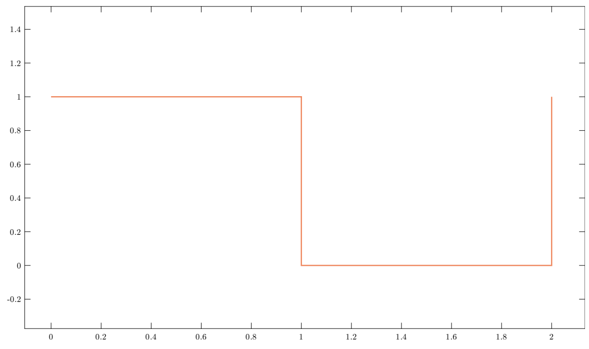
☑ Interpret vector parameters as 1-D

(b)

Figure 4.15. Figure 4.15a gives a snippet from the popup menu of the Parameters of the Pulse Generator Block (located in the upper left in Figure 4.9, called ‘System Trigger’). Users can alter various Block specific Parameters in this menu. Figure 4.15b gives the Block’s Parameters after sanitization: the Parameters ‘Pulse type’, ‘Period’, and ‘Pulse Width’ are affected by the resetting of Parameters. Due to the sanitization, other default Block Parameters may become accessible, like ‘Sample time‘ in this case.



(a)



(b)

Figure 4.16. The exemplary effect of resetting Dialog Parameters: the original Block (Figure 4.15a) produces rapid pulses (Figure 4.16a), while the sanitized version (Figure 4.15b) has a much longer cycle length (Figure 4.16b).

the obvious obfuscation transformations. However, a Block’s Parameters and their values can be inspected in popup menus like the one given in Figure 4.15a. If the user decides to reset the ‘Dialog Parameters’ some of the Block’s Parameters are reset to their default values. For this Block’s reset, the conditional Parameter ‘Sample time’ is revealed in Figure 4.15b. Changing a Block’s Parameters can have a dramatic functional impact, as can be seen for the Pulse Generator Block from Figure 4.15, whose behavior changed from the one shown in Figure 4.16a to the one in Figure 4.16b, after the Parameter reset shown in Figure 4.15.

4.2.5 Evaluation

To ensure that Smoke works as intended and designed, we test its behavior in multiple ways: we test, whether Smoke's obfuscation preserves structural integrity for all its transformations in Section 4.2.5, whether Smoke's obfuscation does not alter a model's behavior in Section 4.2.5, and whether Smoke's sanitization *does alter* model behavior in Section 4.2.5. We further check Smoke's coverage in Section 4.2.5. For all our experiments, we use a diverse set of models, which we briefly introduce in Section 4.2.5.

Experimental Design

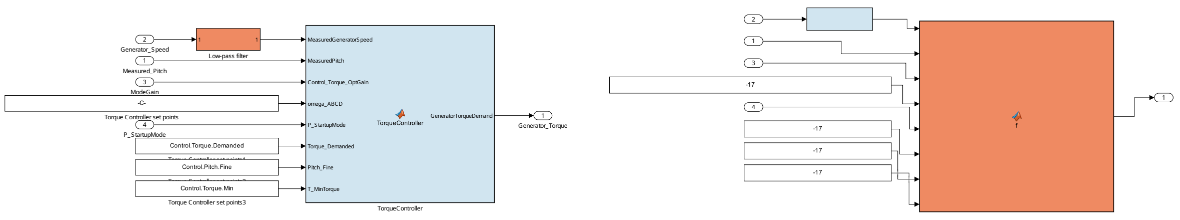
Subjects and Setup As described earlier, Simulink is a vast and diverse ecosystem. To ensure that the various kinds of models and use cases are covered by Smoke, we apply Smoke on the model collection SLNET [337]. This is a comprehensive set of 9,105 open-source models covering multiple domains like electronics, aerospace, robotics, or medicine. The collection encompasses small and big models for various purposes like toy projects, industrial application, etc. SLNET was previously used as a benchmark set in other empirical studies on Simulink models [356, 6]. 8,814 of the models were loadable error-free in our setup of Simulink with MATLAB R2024b on our laptop with Windows11, 96GB RAM, and Intel i9-13980HX processor. Models that were not loadable were not necessarily broken, but typically had some libraries missing that were needed in callbacks of Blocks. Five more models were excluded from our evaluation, as they caused MATLAB crashes during model simulation or model backup – both of these functions are called in our evaluation pipeline. This left 8,809 models for our evaluation. In terms our analysis of these models presented in the remainder of this section, we found more than 160 unique Block types and more than 10,000 unique types of Block Parameters that interact in various, complex ways. Thus, SLNET presents a large set of numerous challenging corner cases for Smoke.

In our evaluation, we applied Smoke sequentially on each SLNET model to obtain a pair of an original model and a transformed model. We then analyzed each pair further as described in the next sections.

Measuring Robustness, Performance, and Structural Integrity In a first step, we measured the robustness of Smoke, *i.e.*, how often Smoke was applicable to our models without error. We also recorded Smoke's runtime performance, *e.g.*, transformed Blocks per second, when applying the complete list of obfuscations and sanitizations that are shown as checked in Figure 4.12.

One of our goals of Smoke is that it does not alter a model's structure, for both sanitization and obfuscation. Our first approach of validating the structural integrity was to employ an existing model comparison tool. However, the built-in tool of Simulink[14] was not able to accurately match pairs of original and transformed models. This was surprising, as it even failed to match easy cases like the one shown in Figure 4.17. There, the Model Comparison tool erroneously matched the peach colored and the aqua colored blocks of this small Subsystem. In our second attempt, we tried the clone detection tools Conqat and

[14] `mathworks.com/help/simulink/model-comparison.html`



(a) (b)

Figure 4.17. The Simulink-internal Model Comparison tool struggles to match even easy cases: here the peach and aqua colored Blocks from the original (Figure 4.17a) and the anonymized model (Figure 4.17b) are erroneously matched.

SIMONE. However, we did not get Conqat to run, and SIMONE is already outdated, as it is only able to handle `.mdl` models.[15]

We thus employed a signature-based model comparison [70], using model metrics for which we developed evaluation scripts specifically for this assessment. It uses simple model metrics like the number of Blocks, Lines, Subsystems, unique Block types of a model, and its cyclomatic complexity. All these metrics were either used previously in the literature [1, 10], or are given as relevant by Simulink researchers themselves [5]. We classify a model to be structurally integral if its signature remained unchanged.

Behavioral (Non-)Integrity of Transformations In regard to the preservation of model behavior, our goal with SMOKE is two-fold: all obfuscation transformations shall *preserve* the original model's behavior, while the sanitization transformations shall *alter* a model's behavior. To test whether a model's behavior is preserved after transformation, we treat models as black boxes into which we feed the same inputs and observe their outputs. If a transformed model shows a different output, given the same input, we classify it as altered behavior. We optimistically classify all obfuscated models, that show the same output as behavior-preserving. This is optimistic in the sense of our setup failing to try a 'deciding' input that produces diverging outputs. All sanitized models that have the same output are inspected, and classified manually, in a second step.

To record the input and output behavior, our evaluation scripts construct a wrapper for each model. The wrapper replaces the model inports with signal generators, and if necessary type converters, and additionally records the values of the outgoing signals, as shown in Figure 4.17. We then simulate both models of a pair and compare their outputs.

Note that we only construct wrappers for models that are actually compilable, and thus could demonstrate any behavior. However, we observed that often (see Table 4.6) the status of compilability is changed from sanitizing it. Each compilability status change is also counted as breaking the model's behavioral integrity, as one of the models can demonstrate behavior, while the other cannot.

[15] Simulink uses `.slx` models by default since 2012.

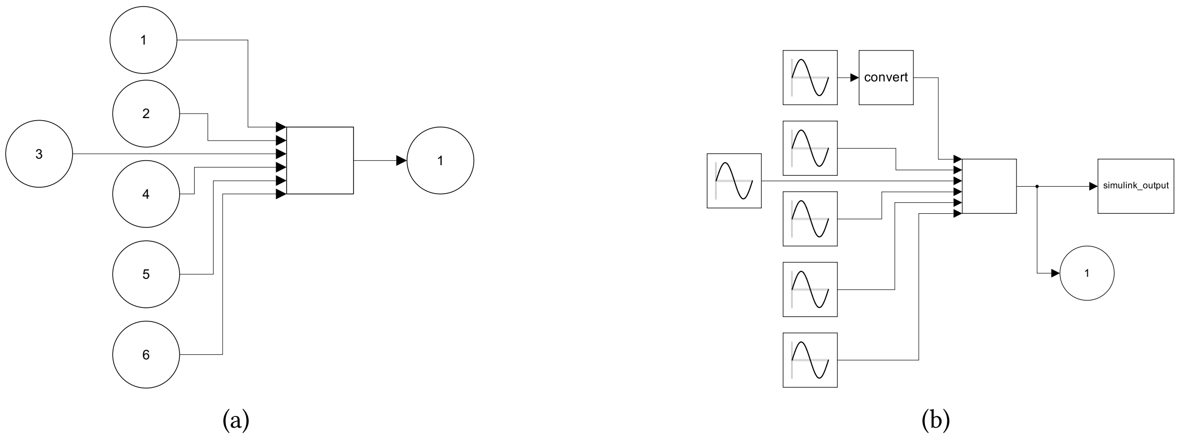


Figure 4.18. Our black box wrapping setup: all model inports in Figure 4.18a are replaced by signal generators in Figure 4.18b. The first port of this example model is of type boolean and a boolean converter is thus added. Similarly, all output signals are tapped into with a To Workspace Block to record the model output.

Results: Robustness, Performance, Structural Integrity

Smoke was able to apply all obfuscations and all sanitizations on all 8,809 models successfully. Smoke worked at a speed of 25.4 Blocks per second, 33.8 Signals per second, 3.5 Subsystems per second, and 0.62 cyclomatic complexity units per second, when all transformations shown in Figure 4.12 are applied. The number of Subsystems showed the highest correlation to Smoke execution time of +0.521, while cyclomatic complexity had the lowest correlation of +0.140.

In all models, the structural integrity was preserved, *i.e.*, no Blocks, Lines, Subsystems were ever added or removed or types of Blocks changed. We found that the cyclomatic complexity changed in 1,954 sanitized models (never in the obfuscated models). This, however stemmed from Simulink's method of calculating cyclomatic complexity: only a compilable model's cyclomatic complexity can be calculated. The compilability of sanitized models, however, switched in many model pairs, as we can see in Table 4.6. We thus ignored cyclomatic complexity for our analysis, and find Smoke to preserve structural integrity for all transformations, as desired.

Behavioral Integrity Results: Obfuscation

As expected, we found all models' behavior to be preserved by Smoke's obfuscation transformations.

Behavioral (Non-)Integrity Results: Sanitization

In a first step of behavioral analysis, we compile all pairs of original and sanitized models – a necessary precursor to their simulation in the next step. Table 4.6 gives a result of the compilation. We can see that most original models are not compilable, already before the sanitization, *i.e.*, $6,132 = 5,933 + 199$ models fail to compile from the start. This is because many of them serve some other purpose, like library models, and are not intended (or impossible) to be compiled, or run. We view non-compilable models as not showing any behavior that could be altered or evaluated automatically, and thus ignore them from the

Table 4.6. Compilation status of models before and after sanitization.

		After Sanitization	
		Compilation Fails	Compilable
Before Sanitization	Compilation Fails	5,933	199
	Compilable	1,638	1,039

further process. The sanitization breaks the compilation of 1,638 models, and interestingly, 199 models become compilable *after* sanitization. This is due to models being in some kind of broken state (in respect to compilation), which sometimes gets fixed by resetting Parameters with Smoke. We view a change in compilability in a model pair as a behavior alteration, as only one of the two models can demonstrate any behavior, while the other cannot. Only 1,039 models are compilable before and after transformation, and these are the models we simulated next.

Table 4.7. Output status of models before and after sanitization.

		After Sanitization	
		Crash/no output	Output
Before Sanitization	Crash/no output	823	27
	Output	31	158

We give the results of the simulation in Table 4.7. Most models produce no output, *i.e.*, they crash in their execution right away, or there was no output to harness in our test setup (*e.g.*, library models). Similar to the compilation, we see 31 models producing no output after the sanitization, and more notably, 27 sanitized models to start producing output. Only 158 models ran their execution crash-free for both model versions. Of these, 16 models produced the exact same output. From our initial set of 8,809 models, we thus automatically classified $8,809 - 16 = 8,793$ models as behaviorally changed, or without behavior.

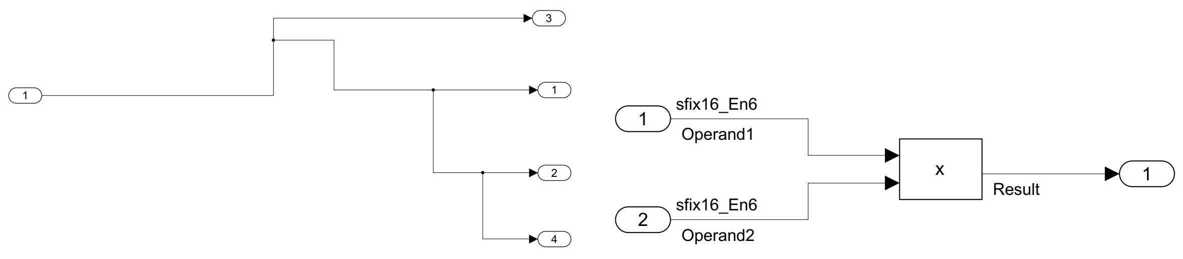
The models that became compilable (199), or executable (27) via the sanitization deserve a closer look. We observed that the sanitization removed custom (but broken) model parts, such as configurations, or it reset data types of Parameter values, removed faulty callbacks, etc. The other way around of sanitized models becoming broken (for compiling 1,638, or running 31) is more obvious: after the sanitization, needed variables, intra-model dependencies, or data types are missing, or Block Parameters become inconsistent with each other.

We further inspected the last 16 stable models to determine why the sanitizing process did not alter their behavior. Our first observation is easily recognizable from Table 4.8: the stable models are much smaller for the various size metrics than their counterparts of the complete SLNET set. Most of their implementations are also completely flat, *i.e.*, they are devoid of any Subsystems. This is intuitive because, on average, smaller models exhibit less complex behavior that could be affected by our sanitizations.

A manual inspection further showed that these models are also simple, probably toy projects. We give two example models in Figure 4.19, where one can see them to be too

Table 4.8. Metric comparison of SLNET models vs. their subset of behaviorally stable models.

		Blocks	Block Types	Signal Lines	Subsystems
mean	SLNET models	133.5	10.8	177.8	18.6
	stable models ⊂ SLNET	7.6	3.4	8.7	0.3
median	SLNET models	30	9	36	4
	stable models ⊂ SLNET	5	3	3	0



(a) (b)

Figure 4.19. Two exemplary models unaffected by sanitizing. They are too small, and or too simple, and Smoke did not change any behavior-affecting Parameters in each.

simple and too small for the sanitization to have any effect. We argue they show no real functionality that needs to be sanitized, *i.e.*, is sensitive for users, in the first place.

In conclusion, Smoke's sanitization altered the behavior of most models, with only a few small and trivial models remaining stable. As we believe models that contain sensitive information are also complex and or big, *i.e.*, containing user-changed Parameters or Functions, we view Smoke's sanitization as effective.

Coverage

In a final step to enhance Smoke's capabilities, we examined the raw model files to improve its coverage, *i.e.*, we checked if Smoke leaves potentially sensitive model elements untouched. In the SLNET model files, we found 160 different Block types and 10,731 different Parameters. These Parameters are either model-wide Parameters or for its various elements, such as Lines, Blocks, Annotation, etc. Many of these Parameters are for internal Simulink use, and users are not supposed to edit them.

Smoke does not support *all* of these Blocks and Parameters, as many of them are not supposed to be changed, or would need individual handling in our implementation, which is not feasible. When designing Smoke, we first came up with a number of obfuscation and sanitization candidates (compare Section 4.2.4). After anonymizing them, we next ran a script that finds model elements that Smoke has thus far left out, but could potentially hold sensitive information. These sensitive parts were then included into transformations of Smoke, and we started the process again. To identify possibly sensitive parts, our script gathered all unique values in all models for each Parameter. We manually went over this list of values and inspected the first, or the first couple of them, to see whether they might hold valuable information. If they did, we updated Smoke so that they are also anonymized. Our heuristic here was to give a closer look at the values of text or numeric type – especially if there were more than 10 different values for a Parameter. Our argument here is that Parameters holding few different values, or even always the same value, are less likely to be sensitive to disclosure. We further inspected a handful of raw transformed model files visually, to see whether any more possibly sensitive information might still be left. Our investigation process does not guarantee that *every possible* sensitive part of models will be removed by Smoke, but it does remove all those that we found.

Threats to Validity

Internal Threats The biggest threat to our internal validity is in our test setup for running the models. Our harnessing to capture outputs (see Figure 4.18) is based on the heuristic that the model inputs and outputs are not hidden somewhere in the model but are on the outer layer. Additionally, some models may need inputs of certain types or values, which our test setup did not try. However, the size of our model pool made a non-model-specific and fast heuristic necessary. Furthermore, our test setup captured outputs for more than a fourth of the models that were compilable (see Table 4.7), while many of the models were never intended to be run in the first place.
Another aspect of our model output analysis in Section 4.2.5 is that we only measured whether the output of a sanitized model differed *in any way* to the original model's output. We cannot sufficiently decide whether "enough" of a model or a model's behavior is altered or removed. Users have to make sure themselves whether a transformed model satisfies their need for information protection.

Regarding the ability to reverse-engineer information or Smoke's coverage (Section 4.2.5): we did not investigate, whether users (or Simulink) hides possibly valuable data in an encrypted way in the model (or raw files). During our investigation of the raw files, we found a few parts that were not intelligible, but we suspect them to be for internal Simulink purposes and thus ignored them. We otherwise completely remove model elements or map Parameter values to the same value. Mathematically, this means that reversing our transformations would require reconstructing the original elements or Parameter values, via a reverse transformation, that would have to 'guess' correctly [59]. Such guessing might be helped, if enough contextual information is still present in the model. We thus caution users of Smoke to anonymize enough, so that the anonymized parts, cannot be inferred from the rest of the model using domain knowledge.

External Threats Although the SLNET evaluation set is extensive and diverse, we have not yet tested Smoke with actual corporate models, which are its intended target. However, prior studies have shown that a subset of models from SLNET is 'industry-like' [1]. While potential industry partners may have additional requests of obfuscating or deleting elements that Smoke currently ignores, such features are easily integrated, as we already demonstrated in Smoke's evolution (compare Sections 4.2.4 and 4.2.5).

4.2.6 Conclusion

With Smoke, we provide a versatile tool that allows users to share Simulink models while protecting their intellectual property. The tool enables selective and fine-grained obfuscation or sanitization of models, all while preserving their structure. Our hope is that SMOKE will enhance collaboration among researchers and industry partners by facilitating the selective protection of models. This will enable secure sharing within the research community and between companies.

We are currently integrating SMOKE as a filter step into a workflow of creating research data management containers [329, 346]. With SMOKE, we ensure sensitive model parts are excluded before they become part of an immutable container. In the future, we hope that Simulink will integrate features of SMOKE natively to make model anonymization even more accessible. SMOKE is open-source[16] and easily extendable, *e.g.*, with additional transformations. We welcome any suggestions for additional features the community would like to see integrated.

[16] `https://github.com/lanpirot/SMOKE`

5 Conclusion

In summary, this dissertation addresses the fundamental challenge of acquiring suitable Simulink models for empirical research – a challenge that has not only hindered progress in model-based engineering but also contributed to a reproduction crisis in the field. While related work has made progress in providing curated datasets, the need for high-quality and industry-representative models remains unmet. This dissertation addresses this gap through three core contributions:

1. (Goal $\mathcal{A}$) We establish the critical need for suitable Simulink models by demonstrating how the scarcity of such models has limited empirical research and worsens reproducibility issues in studies involving Simulink-based systems.
2. (Goal $\mathcal{B}$) We show that empirical research is theoretically and practically feasible despite this challenge by using existing open-source models as valuable subjects for both empirical studies and tool evaluation, even when they do not fully represent industrial best practices.
3. (Goal $\mathcal{C}$) We mitigate the gap with two innovative approaches:
    - GrandSlam, a model synthesizer to generate large-scale Simulink models with controlled properties, for effective fuzzing or scalability experiments.
    - Smoke, a model anonymizer to derive research-ready models from industrial sources, preserving realism while enabling model sharing.

By providing these tools and results, we complement contemporary advancements such as the curation of sophisticated model datasets [337, 352, 10] and advance open science by lowering barriers to entry for researchers. Together, these contributions help bridge the data desert in Simulink research, offering a foundation for future work to expand datasets, refine model generation techniques, and explore new research directions. Ultimately, this work supports the community to conduct more robust, reproducible, and impactful studies, strengthening the empirical rigor and industrial relevance of Simulink research.

List of References

[20] Paul Jaccard. “The Distribution of the Flora in the Alpine Zone”. In: *New Phytologist*. 11. 2. (1912). Pp. 37–50.

[21] Rensis Likert. “A technique for the measurement of attitudes”. In: *Archives of Psychology*. 22. 140. (1932). Pp. 1–55.

[22] Harold W. Kuhn. “The Hungarian Method for the Assignment Problem”. In: *Naval research logistics quarterly*. 2. 1-2. (Mar. 1955). Pp. 83–97. DOI: 10.1002/nav.3800020109.

[23] Jacob Cohen. “A coefficient of agreement for nominal scales”. In: *Educational and psychological measurement*. 20. 1. (Apr. 1960). Pp. 37–46. DOI: 10.1177/001316446002000104.

[24] Jon Louis Bentley. “Multidimensional Binary Search Trees Used for Associative Searching”. In: *Communications of the ACM*. 18. 9. (Sept. 1975). Pp. 509–517. DOI: 10.1145/361002.361007.

[25] Barry W. Boehm, John R. Brown, and Myron Lipow. “Quantitative evaluation of software quality”. In: *Proceedings of the 2nd international conference on Software engineering*. 1976. Pp. 592–605.

[26] Thomas J. McCabe. “A complexity measure”. In: *IEEE Transactions on Software Engineering*. SE-2. 4. (Dec. 1976). Pp. 308–320. DOI: 10.1109/tse.1976.233837.

[27] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. “An Algorithm for Finding Best Matches in Logarithmic Expected Time”. In: *ACM Transactions on Mathematical Software (TOMS)*. 3. 3. (Sept. 1977). Pp. 209–226. DOI: 10.1145/355744.355745.

[28] Fred T. Krogh. “Algorithms Policy”. In: *ACM Transactions on Mathematical Software (TOMS)*. 4. 2. (June 1978). Pp. 97–99. DOI: 10.1145/355780.355781.

[29] Scott N. Woodfield, Hubert E. Dunsmore, and Vincent Y. Shen. “The effect of modularization and comments on program comprehension”. In: *Proceedings of the 5th international conference on Software engineering*. 1981. Pp. 215–223.

[30] James L. Elshoff and Michael Marcotty. “Improving computer program readability to aid modification”. In: *Communications of the ACM*. 25. 8. (Aug. 1982). Pp. 512–521. DOI: 10.1145/358589.358596.

[31] Webb Miller and Eugene W. Myers. “A File Comparison Program”. In: *Software: Practice and Experience*. 15. 11. (Nov. 1985). Pp. 1025–1040. DOI: 10.1002/spe.4380151102.

[32] David F. Williamson, Robert A. Parker, and Juliette S. Kendrick. “The box plot: a simple visual method to interpret data”. In: *Annals of internal medicine.* 110. 11. (June 1989). Pp. 916–921. DOI: 10.7326/0003-4819-110-11-916.

[33] Richard G. Bubenik. “Optimistic computation”. PhD thesis. Rice University, 1990.

[34] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM.* 33. 12. (Dec. 1990). Pp. 32–44. DOI: 10.1145/96267.96279.

[35] Christian Collberg, Clark Thomborson, and Douglas Low. *A Taxonomy of Obfuscating Transformations.* Tech. rep. 148. New Zealand: Department of Computer Science, The University of Auckland, 1997.

[36] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural Computation.* 9. 8. (Nov. 1997). Pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[37] Norman R. Draper and Harry Smith. *Applied Regression Analysis.* 3rd ed. Wiley Series in Probability and Statistics. New York: Wiley-Interscience, 1998, p. 736. ISBN: 9780471170822.

[38] Walter F. Tichy. “Should computer scientists experiment more?” In: *Computer.* 31. 5. (May 1998). Pp. 32–40. DOI: 10.1109/2.675631.

[39] Mike Atallah, Elisa Bertino, Ahmed Elmagarmid, Mohamed Ibrahim, and Vassilios Verykios. “Disclosure limitation of sensitive rules”. In: *Proc. 1999 Workshop on Knowledge and Data Engineering Exchange (KDEX’99).* IEEE Comput. Soc, 1999. Pp. 45–52. DOI: 10.1109/kdex.1999.836532.

[40] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Vol. 463. ACM, 1999.

[41] Victor R. Basili, Forrest Shull, and Filippo Lanubile. “Building knowledge through families of experiments”. In: *IEEE Transactions on Software Engineering.* 25. 4. (1999). Pp. 456–473. DOI: 10.1109/32.799939.

[42] Martin Fowler and P. Becker. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley object technology series. Addison-Wesley, 1999. ISBN: 9780201485677. DOI: 10.1007/3-540-45672-4_31.

[43] Eugene J Webb, Donald T Campbell, Richard D Schwartz, and Lee Sechrest. *Unobtrusive measures.* Vol. 2. Sage Publications, 1999.

[44] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. “On the (Im)possibility of Obfuscating Programs”. In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference.* Berlin/Heidelberg, Germany: Springer, 2001. Ed. by Joe Kilian. Pp. 1–18. DOI: 10.1007/3-540-44647-8_1.

[45] Andreas Rau. “On model-based development: decomposition and data abstraction in Simulink”. In: *Gesellschaft fuer Informatik, FG.* 2. 1. (2001).

[46] Victor R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. *Goal Question Metric (GQM) Approach*. Vol. I. John Wiley & Sons, Jan. 2002. ISBN: 9780471028956. DOI: 10.1002/0471028959.sof142.

[47] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. “Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching”. In: *Proceedings 18th international conference on data engineering*. IEEE Computer Science, 2002. Pp. 117–128. DOI: 10.1109/icde.2002.994702.

[48] Tom Mens. “A State-of-the-Art Survey on Software Merging”. In: *IEEE Transactions on Software Engineering (TSE)*. 28. 5. (May 2002). Pp. 449–462. DOI: 10.1109/tse.2002.1000449.

[49] Mary Shaw. “What makes good research in software engineering?” In: *International Journal on Software Tools for Technology Transfer*. 4. 1. (Oct. 2002). Pp. 1–7. DOI: 10.1007/s10009-002-0083-4.

[50] Eriko Nurvitadhi, Wing Wah Leung, and Curtis Cook. “Do class comments aid Java program understanding?” In: *33rd Annual Frontiers in Education, 2003. FIE 2003.* IEEE, 2003. T3C–T3C. DOI: 10.1109/fie.2003.1263332.

[51] S.R.M. Oliveira and O.R. Zaiane. “Protecting sensitive knowledge by data sanitization”. In: *Third IEEE International Conference on Data Mining*. IEEE Comput. Soc, 2003. Pp. 613–616. DOI: 10.1109/ICDM.2003.1250990.

[52] Rebecca Wirfs-Brock and Alan McKean. *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional, 2003.

[53] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. “A Differencing Algorithm for Object-Oriented Programs”. In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Science, 2004. Pp. 2–13. DOI: 10.1109/ASE.2004.1342719.

[54] Paul Barnard. “Software development principles applied to graphical model development”. In: *AIAA Modeling and Simulation Technologies Conference and Exhibit*. American Institute of Aeronautics and Astronautics, June 2005. P. 5888. DOI: 10.2514/6.2005-5888.

[55] Udo Kelter, Jürgen Wehren, and Jörg Niere. “A Generic Difference Algorithm for UML Models”. In: *Software Engineering*. P-64. (2005). Pp. 105–116.

[56] Jef Raskin. “Comments are More Important than Code: The thorough use of internal documentation is one of the most-overlooked ways of improving software quality and speeding implementation.” In: *Queue*. 3. 2. (Mar. 2005). Pp. 64–65. DOI: 10.1145/1053331.1053354.

[57] Zhenchang Xing and Eleni Stroulia. “UMLDiff: An Algorithm for Object-Oriented Design Differencing”. In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, Nov. 2005. Pp. 54–65. DOI: 10.1145/1101908.1101919.

[58] Irving Copi, Carl Cohen, and Daniel Flage. *Essentials of Logic*. Routledge, July 2006, p. 463. ISBN: 9781315389011. DOI: 10.4324/9781315389028.

[59] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM systems journal*. 45. 3. (2006). Pp. 621–645. DOI: 10.1147/sj.453.0621.

[60] Mehrdad Sabetzadeh and Steve Easterbrook. "View Merging in the Presence of Incompleteness and Inconsistency". In: *Requirements Engineering*. 11. 3. (May 2006). Pp. 174–193. DOI: 10.1007/s00766-006-0032-y.

[61] Mario F. Triola, William Martin Goodman, Richard Law, and Gerry Labute. *Elementary statistics*. Vol. 32. 4. Pearson/Addison-Wesley Reading, MA, Nov. 2006, p. 456. DOI: 10.2307/1270136.

[62] I. Elaine Allen and Christopher A. Seaman. "Likert scales and data analyses". In: *Quality Progress*. 40. 7. (2007). Pp. 64–65.

[63] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction". In: *IEEE Transactions on Software Engineering (TSE)*. 33. 11. (Nov. 2007). Pp. 725–743. DOI: 10.1109/tse.2007.70731.

[64] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap". In: *Future of Software Engineering (FOSE'07)*. IEEE, May 2007. Pp. 37–54. DOI: 10.1109/fose.2007.14.

[65] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. "A survey and taxonomy of approaches for mining software repositories in the context of software evolution". In: *Journal of software maintenance and evolution: Research and practice*. 19. 2. (Mar. 2007). Pp. 77–131. DOI: 10.1002/smr.344.

[66] Miryung Kim, David Notkin, and Dan Grossman. "Automatic Inference of Structural Changes for Matching Across Program Versions". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Science, May 2007. Pp. 333–343. DOI: 10.1109/icse.2007.20.

[67] Barbara Kitchenham and Stuart Charters. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report EBSE-2007-01. University of Keele, 2007.

[68] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. "Matching and Merging of Statecharts Specifications". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Science, May 2007. Pp. 54–64. DOI: 10.1109/icse.2007.50.

[69] Heather A. Piwowar, Roger S. Day, and Douglas B. Fridsma. "Sharing detailed research data is associated with increased citation rate". In: ed. by John Ioannidis. *PloS one*. 2. 3. (Mar. 2007). e308. DOI: 10.1371/journal.pone.0000308.

[70] Petri Selonen and Markus Kettunen. “Metamodel-based inference of inter-model correspondence”. In: *11th European Conference on Software Maintenance and Reengineering (CSMR’07).* IEEE Computer Science, 2007. Pp. 71–80. DOI: 10.1109/csmr.2007.31.

[71] William J. Tastle and Mark J. Wierman. “Consensus and dissention: A measure of ordinal dispersion”. In: *International Journal of Approximate Reasoning.* 45. 3. (Aug. 2007). Pp. 531–545. DOI: 10.1016/j.ijar.2006.06.024. North American Fuzzy Information Processing Society Annual Conference NAFIPS ’2005.

[72] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. “Difference Computation of Large Models”. In: *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE).* ACM, Sept. 2007. Pp. 295–304. DOI: 10.1145/1287624.1287665.

[73] Cédric Brun and Alfonso Pierantonio. “Model Differences in the Eclipse Modeling Framework”. In: *UPGRADE.* 9. 2. (2008).

[74] Marco D’Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. “Analysing software repositories to understand software evolution”. In: *Software evolution.* Springer, 2008, pp. 37–67. ISBN: 9783540764403. DOI: 10.1007/978-3-540-76440-3_3.

[75] Ahmed E. Hassan. “The road ahead for mining software repositories”. In: *Frontiers of Software Maintenance.* IEEE, Sept. 2008. Pp. 48–57. DOI: 10.1109/fosm.2008.4659248.

[76] Reinhold Plösch, Harald Gruber, Anja Hentschel, Christian Körner, Gustav Pomberger, Stefan Schiffer, Matthias Saft, and Stephan Storck. “The EMISQ method and its tool support-expert-based evaluation of internal software quality”. In: *Innovations in Systems and Software Engineering.* 4. 1. (Jan. 2008). Pp. 3–15. DOI: 10.1007/s11334-007-0039-7.

[77] Raymond P. L. Buse and Westley R. Weimer. “Learning a metric for code readability”. In: *IEEE Transactions on software engineering.* 36. 4. (July 2009). Pp. 546–558. DOI: 10.1109/tse.2009.70.

[78] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. “Ldiff: An Enhanced Line Differencing Tool”. In: *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE Computer Science, 2009. Pp. 595–598. DOI: 10.1109/icse.2009.5070564.

[79] Fecir Duran, İsmail Atacak, and Ömer Faruk Bay. “Simulink stateflow for algorithm learning”. In: *Procedia - Social and Behavioral Sciences.* 1. 1. (2009). Pp. 554–558. DOI: 10.1016/j.sbspro.2009.01.100.

[80] Jay Graylin, Joanne E. Hale, Randy K. Smith, Hale David, Nicholas A. Kraft, and Charles Ward. “Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship”. In: *Journal of Software Engineering and Applications.* 2. 03. (2009). P. 137.

[81] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. “Language evolution in practice: The history of GMF”. In: *International Conference on Software Language Engineering (SLE)*. Springer Berlin Heidelberg, 2009. Pp. 3–22. DOI: 10.1007/978-3-642-12107-4_3.

[82] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. “Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing”. In: *2009 ICSE Workshop on Comparison and Versioning of Software Models, 2009*. IEEE Computer Science, May 2009. Pp. 1–6. DOI: 10.1109/cvsm.2009.5071714.

[83] Peter Liggesmeyer and Mario Trapp. “Trends in embedded software engineering”. In: *IEEE software*. 26. 3. (May 2009). Pp. 19–25. DOI: 10.1109/ms.2009.80.

[84] Thilo Mende, Rainer Koschke, and Felix Beckwermert. “An Evaluation of Code Similarity Identification for the Grow-and-Prune Model”. In: *Journal of Software Maintenance and Evolution: Research and Practice*. 21. 2. (Feb. 2009). Pp. 143–169. DOI: 10.1002/smr.402.

[85] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. “Automatic model generation strategies for model transformation testing”. In: *International Conference on Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2009. Pp. 148–164. DOI: 10.1007/978-3-642-02408-5_11.

[86] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework, Second Edition*. Addison-Wesley, 2009. ISBN: 9780321331885. URL: https://sisis.rz.htw-berlin.de/inh2009/12368099.pdf.

[87] Klaas-Jan Stol and Muhammad Ali Babar. “Reporting Empirical Research in Open Source Software: The State of Practice”. In: *Open Source Ecosystems: Diverse Communities Interacting*. Springer Berlin Heidelberg, 2009. Ed. by Cornelia Boldyreff, Kevin Crowston, Björn Lundell, and Anthony I. Wasserman. Pp. 156–169. DOI: 10.1007/978-3-642-02032-2_15.

[88] Margaret-Anne Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael Muller. “How software developers use tagging to support reminding and refinding”. In: *IEEE Transactions on software engineering*. 35. 4. (2009). Pp. 470–483.

[89] E. Chung, C. Jensen, K. Yatani, V. Kuechler, and K. N. Truong. “Sketching and Drawing in the Design of Open Source Software”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)*. IEEE, Sept. 2010. Pp. 195–202. DOI: 10.1109/vlhcc.2010.34.

[90] Natalia Juristo and Omar S. Gómez. “Replication of software engineering experiments”. In: *Empirical software engineering and verification*. Springer Berlin Heidelberg, 2010, pp. 60–88. ISBN: 9783642252310. DOI: 10.1007/978-3-642-25231-0_2.

[91] Ninus Khamis, René Witte, and Juergen Rilling. "Automatic quality assessment of source code comments: the JavadocMiner". In: *Natural Language Processing and Information Systems: 15th International Conference on Applications of Natural Language to Information Systems, NLDB 2010, Cardiff, UK, June 23-25, 2010. Proceedings 15.* Springer Berlin Heidelberg, 2010. Pp. 68–79. DOI: 10.1007/978-3-642-13881-2_7.

[92] Barbara Kitchenham, Rialette Pretorius, David Budgen, O. Pearl Brereton, Mark Turner, Mahmood Niazi, and Stephen Linkman. "Systematic literature reviews in software engineering–a tertiary study". In: *Information and software technology.* 52. 8. (Aug. 2010). Pp. 792–805. DOI: 10.1016/j.infsof.2010.03.006.

[93] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. "Automatic Variation-Point Identification in Function-Block-Based Models". In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE).* ACM, Oct. 2010. Pp. 23–32. DOI: 10.1145/1868294.1868299.

[94] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies". In: *2010 Asia Pacific Software Engineering Conference.* IEEE, Nov. 2010. Pp. 336–345. DOI: 10.1109/APSEC.2010.46.

[95] Alfredo Capozucca, Betty Cheng, Nicolas Guelfi, and Paul Istoan. "OO-SPL Modelling of the Focused Case Study". In: *Comparing Modeling Approaches (CMA).* ACM, 2011.

[96] Alfredo Capozucca, Betty H.C. Cheng, Geri Georg, Nicolas Guelfi, Paul Istoan, and Gunter Mussbacher. *Requirements Definition Document for a Software Product Line of Car Crash Management Systems.* Technical Report CS-11-105. Colorado State University, Department of Computer Science, June 2011. URL: https://www.cs.colostate.edu/TechReports/Reports/2011/tr11-105.pdf.

[97] Moacyr A.G. De Brito, Leonardo P. Sampaio, G. Luigi, Guilherme A. e Melo, and Carlos A. Canesin. "Comparative analysis of MPPT techniques for PV applications". In: *2011 International Conference on Clean Electrical Power (ICCEP).* IEEE, June 2011. Pp. 99–104. DOI: 10.1109/iccep.2011.6036361.

[98] Jessica DeCuir-Gunby, Patricia Marshall, and Allison Mcculloch. "Developing and Using a Codebook for the Analysis of Interview Data: An Example from a Professional Development Research Project". In: *Field Methods Journal.* 23. 2. (May 2011). Pp. 136–155. DOI: 10.1177/1525822x10388468.

[99] Gordon Fraser and Andreas Zeller. "Exploiting Common Object Usage in Test Case Generation". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation.* IEEE, Mar. 2011. Pp. 80–89. DOI: 10.1109/ICST.2011.53.

[100] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. "Empirical Assessment of MDE in Industry". In: *33rd International Conference on Software Engineering (ICSE).* ACM, May 2011. Pp. 471–480. DOI: 10.1145/1985793.1985858.

[101] ISO/IEC. *Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models*. Standard. ISO/IEC 25010:2011. 2011. URL: https://www.iso.org/standard/35733.html (visited on 10/29/2025).

[102] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. “A rule-based approach to the semantic lifting of model differences in the context of model versioning”. In: *26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Nov. 2011. Pp. 163–172. DOI: 10.1109/ase.2011.6100050.

[103] Klaus Krippendorff. *Computing Krippendorff’s alpha-reliability*. Tech. rep. University of Pennsylvania, 2011.

[104] Andreas Svendsen, Òystein Haugen, and Birger Mòller-Pedersen. “Synthesizing software models: generating train station models automatically”. In: *International SDL Forum*. Springer Berlin Heidelberg, 2011. Pp. 38–53. DOI: 10.1007/978-3-642-25264-8_5.

[105] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. ACM, Aug. 2011. Pp. 283–294. DOI: 10.1145/2345156.1993532.

[106] Stefan Feldmann, Julia Fuchs, and Birgit Vogel-Heuser. “Modularity, variant and version management in plant automation–future challenges and state of the art”. In: *Proceedings of the International Design Conference (IDC)*. 2012. Pp. 1689–1698.

[107] Hans-Georg Fill. “Using Obfuscating Transformations for Supporting the Sharing and Analysis of Conceptual Models”. In: *Multikonferenz Wirtschaftsinformatik 2012 - Teilkonferenz Modellierung betrieblicher Informationssysteme*. Braunschweig: GITO Verlag, 2012. Ed. by Susanne Robra-Bissantz and Dirk Mattfeld.

[108] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with Code Fragments”. In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012. Pp. 445–458. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[109] Wei Hu, Tino Loeffler, and Joachim Wegener. “Quality model based on ISO/IEC 9126 for internal quality of MATLAB/Simulink/Stateflow models”. In: *IEEE International Conference on Industrial Technology*. IEEE, Mar. 2012. Pp. 325–330. DOI: 10.1109/icit.2012.6209958.

[110] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. “Understanding model evolution through semantically lifting model differences with SiLift”. In: *28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sept. 2012. Pp. 638–641. DOI: 10.1109/icsm.2012.6405342.

[111] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. “Adaptability of model comparison tools”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, Sept. 2012. Pp. 306–309. DOI: 10.1145/2351676.2351731.

[112] Sascha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*. Tech. rep. Technische Universität Braunschweig, 2012.

[113] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A. Fernandez. "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases". In: *Empirical Software Engineering*. 18. 1. (Jan. 2012). Pp. 89–116. DOI: 10.1007/s10664-012-9196-x.

[114] Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam. "From Verification to Implementation: A Model Translation Tool and a Pacemaker Case Study". In: *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. IEEE, Apr. 2012. Pp. 173–184. DOI: 10.1109/rtas.2012.25.

[115] Yasaman Talaei Rad and Ramtin Jabbari. "Use of Global Consistency Checking for Exploring and Refining Relationships between Distributed Models: A Case Study". Master's Thesis. Blekinge Institute of Technology, School of Computing, 2012.

[116] Robert Reicherdt and Sabine Glesner. "Slicing MATLAB Simulink models". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. New York City, NY, USA: IEEE Computer Society, June 2012. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. Pp. 551–561. DOI: 10.1109/icse.2012.6227161.

[117] Julia Rubin and Marsha Chechik. "Combining Related Products into Product Lines". In: *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2012. Pp. 285–300. DOI: 10.1007/978-3-642-28872-2_20.

[118] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. "Automatic Library Migration for the Generation of Hardware-In-The-Loop Models". In: *Science of Computer Programming (SCP)*. 77. 2. (Feb. 2012). Pp. 83–95. DOI: 10.1016/j.scico.2010.06.005.

[119] Tiago Boldt Sousa. "Dataflow programming concept, languages and applications". In: *Doctoral Symposium on Informatics Engineering*. 2012.

[120] Zlatko Stapić, Eva García López, Antonio García Cabot, Luis de Marcos Ortega, and Vjeran Strahonja. "Performing systematic literature review in software engineering". In: *CECIIS 2012-23rd International Conference*. 2012.

[121] J. Beau W. Webber. "A bi-symmetric log transformation for wide-range data". In: *Measurement Science and Technology*. 24. 2. (Dec. 2012). Pp. 1–3. DOI: 10.1088/0957-0233/24/2/027001.

[122] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012. ISBN: 9783642290442. DOI: 10.1007/978-3-642-29044-2.

[123] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. "LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Science, Sept. 2013. Pp. 230–239. DOI: 10.1109/icsm.2013.34.

[124] Omar Badreddin, Timothy C. Lethbridge, and Maged Elassar. “Modeling Practices in Open Source Software”. In: *Open Source Software: Quality Verification (OSS).* Springer Berlin Heidelberg, 2013. Ed. by Etiel Petrinja, Giancarlo Succi, Nabil El Ioini, and Alberto Sillitti. Pp. 127–139. DOI: 10.1007/978-3-642-38928-3_9.

[125] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. “A Survey of Variability Modeling in Industrial Practice”. In: *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS).* ACM, Jan. 2013. 7:1–7:8. DOI: 10.1145/2430502.2430513.

[126] James R. Cordy. “Submodel pattern extraction for simulink models”. In: *Proceedings of the 17th International Software Product Line Conference.* Tokyo, Japan: ACM, Aug. 2013. Pp. 7–10. DOI: 10.1145/2491627.2492153.

[127] Yanja Dajsuren, Mark G.J. van den Brand, Alexander Serebrenik, and Serguei Roubtsov. “Simulink Models Are Also Software: Modularity Assessment”. In: *9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA).* ACM, June 2013. Pp. 99–106. DOI: 10.1145/2465478.2465482.

[128] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. “An Exploratory Study of Cloning in Industrial Software Product Lines”. In: *2013 17th European Conference on Software Maintenance and Reengineering.* IEEE Computer Science, Mar. 2013. Pp. 25–34. DOI: 10.1109/csmr.2013.13.

[129] Frank Elberzhager, Alla Rosbach, and Thomas Bauer. “Analysis and Testing of Matlab Simulink Models: A Systematic Mapping Study”. In: *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation.* Lugano, Switzerland: ACM, July 2013. Pp. 29–34. DOI: 10.1145/2489280.2489285.

[130] Benjamin Klatt and Martin Küster. “Improving Product Copy Consolidation by Architecture-Aware Difference Analysis”. In: *QoSA ’13: Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures.* ACM, June 2013. Pp. 117–122. DOI: 10.1145/2465478.2465495.

[131] Benjamin Klatt, Martin Küster, and Klaus Krogmann. “A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies”. In: *Proceedings of the International Workshop on Reverse Variability Engineering (REVE).* 2013. Pp. 1–8.

[132] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, and Yann-Gael Gueheneuc. “Madmatch: Many-to-many approximate diagram matching for design comparison”. In: *IEEE Transactions on Software Engineering (TSE).* 39. 8. (Aug. 2013). Pp. 1090–1111. DOI: 10.1109/tse.2013.9.

[133] M. Petre. “UML in practice”. In: *International Conference on Software Engineering (ICSE).* IEEE, May 2013. Pp. 722–731. DOI: 10.1109/icse.2013.6606618.

[134] Heather A. Piwowar and Todd J. Vision. “Data reuse and the open data citation advantage”. In: *PeerJ.* 1. (Apr. 2013). e175. DOI: 10.7287/peerj.preprints.1v1.

[135] Julia Rubin and Marsha Chechik. "N-Way Model Merging". In: *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*. ACM, Aug. 2013. Pp. 301–311. DOI: 10.1145/2491411.2491446.

[136] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. "Managing Cloned Variants: A Framework and Experience". In: *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, Aug. 2013. Pp. 101–110. DOI: 10.1145/2491627.2491644.

[137] Patricia M. Shields and Nandhini Rangarajan. *A Playbook for Research Methods: Integrating Conceptual Frameworks and Project Management*. New Forums Press, July 2013, p. 292.

[138] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. "Quality analysis of source code comments". In: *2013 21st international conference on program comprehension (icpc)*. IEEE, May 2013. Pp. 83–92. DOI: 10.1109/icpc.2013.6613836.

[139] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[140] Ludo Waltman and Nees Jan van Eck. "A systematic empirical comparison of different approaches for normalizing citation impact indicators". In: *Journal of Informetrics*. 7. 4. (Oct. 2013). Pp. 833–849. DOI: 10.1016/j.joi.2013.08.002.

[141] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. "Industrial adoption of model-driven engineering: Are the tools really the problem?" In: *International Conference on Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2013. Pp. 1–17. DOI: 10.1007/978-3-642-41533-3_1.

[142] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. "Statistical analysis of changes for synthesizing realistic test models". In: *Software Engineering*. (2013).

[143] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The oracle problem in software testing: A survey". In: *IEEE transactions on software engineering*. 41. 5. (May 2014). Pp. 507–525. DOI: 10.1109/tse.2014.2372785.

[144] Håkan Burden, Rogardt Heldal, and Jon Whittle. "Comparing and Contrasting Model-Driven Engineering at Three Large Companies". In: *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, Sept. 2014. Pp. 1–10. DOI: 10.1145/2652524.2652527.

[145] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques". In: *Empirical Software Engineering*. 19. (Feb. 2014). Pp. 1040–1074. DOI: 10.1007/s10664-013-9248-x.

[146] W. Ding, P. Liang, A. Tang, H. v. Vliet, and M. Shahin. "How Do Open Source Communities Document Software Architecture: An Exploratory Survey". In: *19th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, Aug. 2014. Pp. 136–145. DOI: 10.1109/iceccs.2014.26.

[147] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. “Fine-Grained and Accurate Source Code Differencing”. In: *Proceedings of the International Conference on Automated Software Engineering (ASE).* ACM, Sept. 2014. Pp. 313–324. DOI: 10.1145/2642937.2642982.

[148] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. “Benchmarks for model transformations and conformance checking”. In: *1st International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH).* 2014.

[149] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. “The promises and perils of mining GitHub”. In: *11th Working Conference on Mining Software Repositories (MSR).* ACM, May 2014. Pp. 92–101. DOI: 10.1145/2597073.2597074.

[150] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. “Variability Mining: Consistent Semiautomatic Detection of Product-Line Features”. In: *IEEE Transactions on Software Engineering.* 40. 1. (Jan. 2014). Pp. 67–82. DOI: 10.1109/tse.2013.45.

[151] Philip Langer, Tanja Mayerhofer, Manuel Wimmer, and Gerti Kappel. “On the usage of UML: Initial results of analyzing open UML models”. In: *Modellierung 2014.* (2014). Pp. 289–304.

[152] Vu Le, Mehrdad Afshari, and Zhendong Su. “Compiler validation via equivalence modulo inputs”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* Edinburgh, United Kingdom: ACM, June 2014. Pp. 216–226. DOI: 10.1145/2594291.2594334.

[153] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. “Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain”. In: *Model-Driven Engineering Languages and Systems.* Cham: Springer International Publishing, 2014. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran. Pp. 166–182. DOI: 10.1007/978-3-319-11653-2_11.

[154] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. “On the comprehension of program comprehension”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM).* 23. 4. (Sept. 2014). Pp. 1–37. DOI: 10.1145/2622669.

[155] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. “The Relevance of Model-Driven Engineering Thirty Years from Now”. In: *Model-Driven Engineering Languages and Systems (MODELS).* Springer International Publishing, 2014. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran. Pp. 183–200. DOI: 10.1007/978-3-319-11653-2_12.

[156] Thomas Schmorleiz and Ralf Lämmel. “Similarity Management via History Annotation”. In: *Proc. Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*. Dipartimento di Informatica Università degli Studi dell’Aquila, L’Aquila, Italy, 2014. Pp. 45–48.

[157] M. Stephan, M. H. Alalfi, and J. R. Cordy. “Towards a Taxonomy for Simulink Model Mutations”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, Mar. 2014. Pp. 206–215. DOI: 10.1109/icstw.2014.17.

[158] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit*. Tech. rep. Institute of Automation and Information Systems, Technische Universität München, 2014.

[159] Jens Weiland and Peter Manhart. “A classification of modeling variability in Simulink”. In: *Proceedings of the 8th international workshop on variability modelling of software-intensive systems*. ACM, Jan. 2014. Pp. 1–8. DOI: 10.1145/2556624.2556632.

[160] Sven Wenzel. “Unique identification of elements in evolving software models”. In: *Software & Systems Modeling*. 13. 2. (Feb. 2014). Pp. 679–711. DOI: 10.1007/s10270-012-0311-7.

[161] Michael W. Whalen, Anitha Murugesan, Sanjai Rayadurgam, and Mats P.E. Heimdahl. “Structuring Simulink models for verification and reuse”. In: *Proceedings of the 6th International Workshop on Modeling in Software Engineering*. ACM, June 2014. Pp. 19–24. DOI: 10.1145/2593770.2593776.

[162] Tuğrul Yazar. “Design of Dataflow”. In: *Nexus Network Journal*. 17. 1. (Dec. 2014). Pp. 311–325. DOI: 10.1007/s00004-014-0222-8.

[163] Hamed Shariat Yazdi, Mahnaz Mirbolouki, Pit Pietsch, Timo Kehrer, and Udo Kelter. “Analysis and prediction of design model evolution using time series”. In: *International Conference on Advanced Information Systems Engineering*. Springer International Publishing, 2014. Pp. 1–15. DOI: 10.1007/978-3-319-07869-4_1.

[164] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. “Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, Mar. 2014. Pp. 1064–1071. DOI: 10.1145/2554850.2554874.

[165] Pontus Boström and Jonatan Wiik. “Contract-based verification of discrete-time multi-rate Simulink models”. In: *Software & Systems Modeling*. 15. 4. (June 2015). Pp. 1141–1161. DOI: 10.1007/s10270-015-0477-x.

[166] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. “Reasoning About Product-Line Evolution Using Complex Feature Model Differences”. In: *Automated Software Engineering*. 23. 4. (Oct. 2015). Pp. 687–733. DOI: 10.1007/s10515-015-0185-3.

[167] Yanja Dajsuren. "On the design of an architecture framework and quality evaluation for automotive software systems". PhD thesis. Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2015.

[168] Slawomir Duszynski. "Analyzing Similarity of Cloned Software Variants Using Hierarchical Set Models". PhD thesis. University of Kaiserslautern, 2015.

[169] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. "Boa: Ultra-large-scale software repository and source-code mining". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 25. 1. (Dec. 2015). Pp. 1–34. DOI: 10.1145/2803171.

[170] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. "The ECCO Tool: Extraction and Composition for Clone-and-Own". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Science, May 2015. Pp. 665–668. DOI: 10.1109/icse.2015.218.

[171] Juan Gallego-Calderon and Anand Natarajan. "Assessment of wind turbine drive-train fatigue loads under torsional excitation". In: *Engineering Structures*. 103. (Nov. 2015). Pp. 189–202. DOI: 10.1016/j.engstruct.2015.09.008.

[172] Thomas Gerlitz, Quang Minh Tran, and Christian Dziobek. "Detection and Handling of Model Smells for MATLAB/Simulink models". In: *MASE@ MoDELS*. 2015. Pp. 13–22.

[173] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. "SyLVaaS: System Level Formal Verification as a Service". In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, Mar. 2015. Pp. 476–483. DOI: 10.1109/PDP.2015.119.

[174] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach". In: *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, July 2015. Pp. 101–110. DOI: 10.1145/2791060.2791086.

[175] Muhammad Rashid, Muhammad Waseem Anwar, and Aamir M. Khan. "Toward the tools selection in model based system engineering for embedded systems—A systematic literature review". In: *Journal of Systems and Software*. 106. (Aug. 2015). Pp. 150–163. DOI: 10.1016/j.jss.2015.04.089.

[176] Thomas Schmorleiz. "An Annotation-Centric Approach to Similarity Management". Master's Thesis. Universität Koblenz-Landau, 2015.

[177] Matthew Stephan and James R. Cordy. "Identification of Simulink model antipattern instances using model clone detection". In: *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*. New York City, NY, USA: IEEE Computer Society, Sept. 2015. Ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed. Pp. 276–285. DOI: 10.1109/models.2015.7338258.

[178] Matthew Stephan and James 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*. Florence, Italy: IEEE Press, May 2015. Pp. 48–53. doi: 10.1109/mise.2015.16.

[179] Thomas Strathmann and Jens Oehlerking. “Verifying Properties of an Electro-Mechanical Braking System”. In: *2nd Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH 2015)*. EasyChair, Apr. 2015. Pp. 49–40. doi: 10.29007/x87p.

[180] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. “When and Why Your Code Starts to Smell Bad”. In: *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. IEEE, May 2015. Pp. 403–414. doi: 10.1109/ICSE.2015.59.

[181] Ken Vanherpen, Joachim Denil, Hans Vangheluwe, and Paul De Meulenaere. “Model transformations for round-trip engineering in control deployment co-design”. In: *SpringSim (TMS-DEVS)*. 920. (2015). Pp. 55–62.

[182] Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer, and Matthias Tichy. “Evolution of Software in Automated Production Systems: Challenges and Research Directions”. In: *Journal of Systems and Software (JSS)*. 110. (Dec. 2015). Pp. 54–84. doi: 10.1016/j.jss.2015.08.026.

[183] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. “Synthesizing realistic test models”. In: *Computer Science-Research and Development*. 30. 3-4. (June 2015). Pp. 231–253. doi: 10.1007/s00450-014-0255-y.

[184] Önder Babur. “Statistical Analysis of Large Sets of Models”. In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, Aug. 2016. Pp. 888–891. doi: 10.1145/2970276.2975938.

[185] Mário André de F. Farias, Renato Novais, Methanias Colaço Júnior, Luís Paulo da Silva Carvalho, Manoel Mendonça, and Rodrigo Oliveira Spínola. “A systematic mapping study on mining software repositories”. In: *31st Annual ACM Symposium on Applied Computing*. ACM, Apr. 2016. Pp. 1472–1479. doi: 10.1145/2851613.2851786.

[186] T. Gerlitz and S. Kowalewski. “Flow Sensitive Slicing for MATLAB/Simulink Models”. In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, Apr. 2016. Pp. 81–90. doi: 10.1109/WICSA.2016.23.

[187] Xiao He, Wenfeng Li, Tian Zhang, and Yi Liu. “Towards Parallel Model Generation for Random Performance Testing of Model-Oriented Operations”. In: *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, July 2016. Pp. 57–64. doi: 10.1109/TASE.2016.26.

[188] Xiao He, Tian Zhang, Chang-Jun Hu, Zhiyi Ma, and Weizhong Shao. “An MDE performance testing framework based on random model generation”. In: *Journal of Systems and Software*. 121. (Nov. 2016). Pp. 247–264. doi: 10.1016/j.jss.2016.04.044.

[189] Regina Hebig, Truong Ho Quang, Michel R.V. Chaudron, Gregorio Robles, and Miguel Angel Fernandez. “The quest for open source projects that use UML: mining GitHub”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, Oct. 2016. Pp. 173–183. DOI: 10.1145/2976767.2976778.

[190] D. Holling, A. Hofbauer, A. Pretschner, and M. Gemmar. “Profiting from Unit Tests for Integration Testing”. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2016. Pp. 353–363. DOI: 10.1109/ICST.2016.28.

[191] Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Qing Xie, Sangmin Park, Kunal Taneja, and B. M. Mainul Hossain. “RUGRAT: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications”. In: *Software: Practice and Experience*. 46. 3. (2016). Pp. 405–431. DOI: https://doi.org/10.1002/spe.2290.

[192] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. “Summarizing source code using a neural attention model”. In: *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2016. Pp. 2073–2083. DOI: 10.18653/v1/p16-1195.

[193] Yuta Kuroki, Myungryun Yoo, and Takanori Yokoyama. “A Simulink to UML model transformation tool for embedded control software development”. In: *IEEE International Conference on Industrial Technology, ICIT 2016, Taipei, Taiwan, March 14-17, 2016*. IEEE, Mar. 2016. Pp. 700–706. DOI: 10.1109/ICIT.2016.7474835.

[194] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. “Automated Test Suite Generation for Time-Continuous Simulink Models”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. ACM, May 2016. Pp. 595–606. DOI: 10.1145/2884781.2884797.

[195] Amina Ahmed Nacer, Elio Goettelmann, Samir Youcef, Abdelkamel Tari, and Claude Godart. “Obfuscating a business process by splitting its logic with fake fragments for securing a multi-cloud deployment”. In: *2016 IEEE World Congress on Services (SERVICES)*. IEEE, June 2016. Pp. 18–25. DOI: 10.1109/services.2016.9.

[196] Marta Olszewska, Yanja Dajsuren, Harald Altinger, Alexander Serebrenik, Marina Waldén, and Mark G.J. van den Brand. “Tailoring complexity metrics for Simulink models”. In: *Proccedings of the 10th European Conference on Software Architecture Workshops*. New York, NY, United States: ACM, Nov. 2016. Pp. 1–7. DOI: 10.1145/2993412.3004853.

[197] Jan Schroeder, Christian Berger, Miroslaw Staron, Thomas Herpel, and Alessia Knauss. “Unveiling anomalies and their impact on software quality in model-based automotive software revisions with software metrics and domain experts”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, July 2016. Pp. 154–164. DOI: 10.1145/2931037.2931060.

[198] Christoph Daniel Schulze, Christina Plöger, and Reinhard von Hanxleden. “On comments in visual languages”. In: *Diagrammatic Representation and Inference: 9th International Conference, Diagrams 2016, Philadelphia, PA, USA, August 7-10, 2016, Proceedings 9*. 2016. Pp. 219–225.

[199] Rodrigo Silva and Francila Neiva. *Systematic Literature Review in Computer Science - A Practical Guide*. en. Technical Report. ResearchGate, Nov. 2016. DOI: 10.13140/RG.2.2.35453.87524.

[200] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. “RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules”. In: *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2016. Pp. 122–140. DOI: 10.1007/978-3-662-49665-7_8.

[201] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Scientific data*. 3. 1. (Mar. 2016). Pp. 1–9. DOI: 10.1038/sdata.2016.18.

[202] David Wille, Sandro Schulze, and Ina Schaefer. “Variability Mining of State Charts”. In: *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*. ACM, Oct. 2016. Pp. 63–73. DOI: 10.1145/3001867.3001875.

[203] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. “Custom-Tailored Variability Mining for Block-Based Languages”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Science, Mar. 2016. Pp. 271–282. DOI: 10.1109/saner.2016.13.

[204] Hamed Shariat Yazdi, Lefteris Angelis, Timo Kehrer, and Udo Kelter. “A framework for capturing, statistically modeling and analyzing the evolution of software models”. In: *Journal of Systems and Software*. 118. (Aug. 2016). Pp. 176–207. DOI: 10.1016/j.jss.2016.05.010.

[205] Silvia Abrahão, Francis Bourdeleau, Betty Cheng, Sahar Kokaly, Richard Paige, Harald Stöerrle, and Jon Whittle. “User Experience for Model-Driven Engineering: Challenges and Future Directions”. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, Sept. 2017. Pp. 229–236. DOI: 10.1109/MODELS.2017.5.

[206] Önder Babur and Loek Cleophas. “Using N-Grams for the Automated Clustering of Structural Models”. In: *SOFSEM 2017: Theory and Practice of Computer Science*. Springer, 2017. Pp. 510–524. DOI: 10.1007/978-3-319-51963-0_40.

[207] Alessio Balsini, Marco Di Natale, Marco Celia, and Vassilios Tsachouridis. “Generation of simulink monitors for control applications from formal requirements”. In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2017. Pp. 1–9. DOI: 10.1109/SIES.2017.7993389.

[208] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. “Component and Connector Views in Practice: An Experience Report”. In: *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems.* Austin, Texas: IEEE Press, Sept. 2017. Pp. 167–177. DOI: 10.1109/MODELS.2017.29.

[209] Hamza Bourbouh, Pierre-Loic Garoche, Christophe Garion, Arie Gurfinkel, Temesghen Kahsai, and Xavier Thirioux. “Automated analysis of Stateflow models”. In: *21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2017).* EasyChair, 2017. Pp. 144–161. DOI: 10.29007/b8gq.

[210] Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. “A Synchronous Look at the Simulink Standard Library”. In: *ACM Trans. Embed. Comput. Syst.* 16. 5s. (Sept. 2017). Pp. 1–24. DOI: 10.1145/3126516.

[211] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 2017. ISBN: 9783031025495. DOI: 10.1007/978-3-031-02549-5.

[212] Shafiul Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner. “CyFuzz: A Differential Testing Framework for Cyber-Physical Systems Development Environments”. In: *Cyber Physical Systems. Design, Modeling, and Evaluation.* Cham: Springer International Publishing, 2017. Ed. by Christian Berger, Mohammad Reza Mousavi, and Rafael Wisniewski. Pp. 46–60. DOI: 10.1007/978-3-319-51738-4_4.

[213] V. Cosentino, J. L. Cánovas Izquierdo, and J. Cabot. “A Systematic Mapping Study of Software Development With GitHub”. In: *IEEE Access.* 5. (2017). Pp. 7173–7192. DOI: 10.1109/access.2017.2682323.

[214] Norman K. Denzin. *Sociological Methods.* Routledge, July 2017. ISBN: 9781315129945. DOI: 10.4324/9781315129945.

[215] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. “Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line”. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE Computer Science, Feb. 2017. Pp. 316–326. DOI: 10.1109/saner.2017.7884632.

[216] Nikhil Gupta, Fei Chen, Nektarios Georgios Tsoutsos, and Michail Maniatakos. “ObfusCADe: Obfuscating Additive Manufacturing CAD Models Against Counterfeiting: Invited”. In: *Proceedings of the 54th Annual Design Automation Conference 2017.* Austin, TX, USA: ACM, June 2017. Pp. 1–6. DOI: 10.1145/3061639.3079847.

[217] Alireza Haghighatkhah, Ahmad Banijamali, Olli-Pekka Pakanen, Markku Oivo, and Pasi Kuvaja. “Automotive software engineering: A systematic mapping study”. In: *Journal of Systems and Software.* 128. (June 2017). Pp. 25–55. DOI: 10.1016/j.jss.2017.03.005.

[218] Zhenying Jiang, Xiao Wu, Zeqian Dong, and Ming Mu. “Optimal Test Case Generation for Simulink Models Using Slicing”. In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, July 2017. Pp. 363–369. DOI: 10.1109/QRS-C.2017.67.

[219] Guido Lobrano. *Making Sense of Competition Law Compliance for, A practical guide for SMEs*. Brussels, Belgium: Business Europe, 2017.

[220] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. “ESPLA: A Catalog of Extractive SPL Adoption Case Studies”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, Sept. 2017. Pp. 38–41. DOI: 10.1145/3109729.3109748.

[221] Paola Masuzzo and Lennart Martens. *Do you speak open science? Resources and tips to learn the language*. Tech. rep. PeerJ Preprints, Jan. 2017. DOI: 10.7287/peerj.preprints.2689v1.

[222] Vu Trieu Minh, Abouelkheir Moustafa, and Mart Tamre. “Design and simulations of dual clutch transmission for hybrid electric vehicles”. In: *International Journal of Electric and Hybrid Vehicles*. 9. 4. (2017). Pp. 302–321. DOI: 10.1504/ijehv.2017.089873.

[223] A. Morozov, K. Ding, T. Chen, and K. Janschek. “Test Suite Prioritization for Efficient Regression Testing of Model-Based Automotive Software”. In: *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, Nov. 2017. Pp. 20–29. DOI: 10.1109/SATE.2017.11.

[224] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. “Curating GitHub for engineered software projects”. In: *Empir. Softw. Eng.* 22. 6. (Dec. 2017). Pp. 3219–3253. DOI: 10.7287/peerj.preprints.2617v1.

[225] P. Norouzi, Ö. C. Kıvanç, and Ö. Üstün. “High performance position control of double sided air core linear brushless DC motor”. In: *2017 10th International Conference on Electrical and Electronics Engineering (ELECO)*. Nov. 2017. Pp. 233–238.

[226] Luca Pascarella and Alberto Bacchelli. “Classifying Code Comments in Java Open-Source Software Systems”. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2017. Pp. 227–237. DOI: 10.1109/MSR.2017.63.

[227] Truong Ho-Quang, Regina Hebig, Gregorio Robles, Michel RV Chaudron, and Miguel Angel Fernandez. “Practices and perceptions of UML use in open source projects”. In: *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, May 2017. Pp. 203–212. DOI: 10.1109/icse-seip.2017.28.

[228] A. C. Rao, A. Raouf, G. Dhadyalla, and V. Pasupuleti. “Mutation Testing Based Evaluation of Formal Verification Tools”. In: *2017 International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, Oct. 2017. Pp. 1–7. DOI: 10.1109/DSA.2017.10.

[229] Daniel F. Savarese. *libssrckdtree-j*. Source Code. Available online at https://www.savarese.com/software/libssrckdtree-j/. 2017. (Visited on 10/29/2025).

[230] Alexander Schlie, David Wille, Sandro Schulze, Loek Cleophas, and Ina 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, SPLC 2017, Volume A, Sevilla, Spain, September 25-29, 2017.* New York City, NY, USA: ACM, Sept. 2017. Ed. by Myra B. Cohen, Mathieu Acher, Lidia Fuentes, Daniel Schall, Jan Bosch, Rafael Capilla, Ebrahim Bagheri, Yingfei Xiong, Javier Troya, Antonio Ruiz Cortés, and David Benavides. Pp. 215–224. DOI: 10.1145/3106195.3106225.

[231] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2017. Pp. 579–594. DOI: 10.1109/SP.2017.23.

[232] Bilal Amir and Paul Ralph. "There is No Random Sampling in Software Engineering Research". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. Gothenburg, Sweden: ACM, May 2018. Pp. 344–345. DOI: 10.1145/3183440.3195001.

[233] Aitor Arrieta, Shuai Wang, Ainhoa Arruabarrena, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. "Multi-Objective Black-Box Test Case Selection for Cost-Effectively Testing Simulation Models". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Kyoto, Japan: ACM, July 2018. Pp. 1411–1418. DOI: 10.1145/3205455.3205490.

[234] S. A. Chowdhury. "Understanding and Improving Cyber-Physical System Models and Development Tools". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. ACM, May 2018. Pp. 452–453. DOI: 10.1145/3183440.3183455.

[235] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner. "Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain With SLforge". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden: ACM, May 2018. Pp. 981–992. DOI: 10.1145/3180155.3180231.

[236] Shafiul Azam Chowdhury, Lina Sera Varghese, Soumik Mohian, Taylor T Johnson, and Christoph Csallner. "A curated corpus of Simulink models for model-based empirical studies". In: *2018 IEEE/ACM 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*. Gothenburg, Sweden: ACM, May 2018. Pp. 45–48. DOI: 10.1145/3196478.3196484.

[237] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. "Compiler fuzzing through deep learning". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Amsterdam, Netherlands: ACM, July 2018. Pp. 95–105. DOI: 10.1145/3213846.3213848.

[238] Heiko Doerr and Ferry Bachmann. *Analysis and Improvement of Model Architectures for Safety Related Systems.* Tech. rep. SAE Technical Paper, Apr. 2018. DOI: 10.4271/2018-01-1077.

[239] Sinem Getir, Lars Grunske, André van Hoorn, Timo Kehrer, Yannic Noller, and Matthias Tichy. "Supporting semi-automatic co-evolution of architecture and fault tree models". In: *Journal of Systems and Software.* 142. (Aug. 2018). Pp. 115–135. DOI: 10.1016/j.jss.2018.04.001.

[240] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. "Deep code comment generation". In: *Proceedings of the 26th conference on program comprehension.* ACM, May 2018. Pp. 200–210. DOI: 10.1145/3196321.3196334.

[241] A. Khelifi, N. M. Ben Lakhal, H. Gharsallaoui, and O. Nasri. "Artificial Neural Network-based Fault Detection". In: *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT).* IEEE, Apr. 2018. Pp. 1017–1022. DOI: 10.1109/CoDIT.2018.8394963.

[242] Harold Klee and Randal Allen. *Simulation of Dynamic Systems with MATLAB and Simulink.* 3rd Ed. Boca Raton, FL, USA: CRC Press, Taylor & Francis Group, 2018. ISBN: 9781315154176. DOI: 10.1201/9781315154176.

[243] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. "Apo-Games: A Case Study for Reverse Engineering Variability from Cloned Java Variants". In: *Proceedings of the International Software Product Line Conference (SPLC).* ACM, Sept. 2018. Pp. 251–256. DOI: 10.1145/3233027.3236403.

[244] Evgeny Kusmenko, Igor Shumeiko, Bernhard Rumpe, and Michael von Wenckstern. "Fast Simulation Preorder Algorithm". In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development.* Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2018. Pp. 256–267. DOI: 10.5220/0006722102560267.

[245] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. "Variability extraction and modeling for product variants". In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1.* Gothenburg, Sweden: ACM, Sept. 2018. P. 250. DOI: 10.1145/3233027.3236396.

[246] Rajib Mall. *Fundamentals of Software Engineering.* PHI Learning Pvt. Ltd., 2018.

[247] Vera Pantelic, Steven Postma, Mark Lawford, Monika Jaskolka, Bennett Mackenzie, Alexandre Korobkine, Marc Bender, Jeff Ong, Gordon Marks, and Alan Wassyng. "Software engineering practices and Simulink: bridging the gap". In: *International Journal on Software Tools for Technology Transfer.* 20. 1. (Mar. 2018). Pp. 95–117. DOI: 10.1007/s10009-017-0450-9.

[248] Asma Rebaya, Kaouther Gasmi, and Salem Hasnaoui. "A Simulink-Based Rapid Prototyping Workflow for Optimizing Software/Hardware Programming". In: *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM).* IEEE, Sept. 2018. Pp. 1–6. DOI: 10.23919/softcom.2018.8555777.

[249] Alexander Schaap, Gordon Marks, Vera Pantelic, Mark Lawford, Gehan Selim, Alan Wassyng, and Lucian Patcas. "Documenting Simulink Designs of Embedded Systems". In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. Copenhagen, Denmark: ACM, Oct. 2018. Pp. 47–51. DOI: 10.1145/3270112.3270115.

[250] Alexander Schaap, Gordon Marks, Vera Pantelic, Mark Lawford, Gehan Selim, Alan Wassyng, and Lucian Patcas. "Documenting Simulink designs of embedded systems". In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. Copenhagen, Denmark: Association for Computing Machinery, 2018. Pp. 47–51. DOI: 10.1145/3270112.3270115.

[251] Alexander Schlie, Sandro Schulze, and Ina Schaefer. "Comparing Multiple MATLAB/Simulink Models Using Static Connectivity Matrix Analysis". In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. New York City, NY, USA: IEEE Computer Society, Sept. 2018. Pp. 160–171. DOI: 10.1109/icsme.2018.00026.

[252] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. "A graph solver for the automated generation of consistent domain-specific models". In: *40th International Conference on Software Engineering (ICSE)*. ACM, May 2018. Pp. 969–980. DOI: 10.1145/3180155.3180186.

[253] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. "Pydriller: Python framework for mining software repositories". In: *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Oct. 2018. Pp. 908–911. DOI: 10.1145/3236024.3264598.

[254] Klaas-Jan Stol and Brian Fitzgerald. "The ABC of Software Engineering Research". In: *ACM Trans. Softw. Eng. Methodol.* 27. 3. (Sept. 2018). Pp. 1–51. DOI: 10.1145/3241743.

[255] Harald Störrle. "On the impact of size to the understanding of UML diagrams". In: *Software and Systems Modeling (SoSyM)*. 17. 1. (May 2018). Pp. 115–134. DOI: 10.1007/s10270-016-0529-x.

[256] David Wille, Önder Babur, Loek Cleophas, Christoph Seidl, Mark van den Brand, and Ina Schaefer. "Improving custom-tailored variability mining using outlier and cluster detection". In: *Science of Computer Programming*. 163. (Oct. 2018). Pp. 62–84. DOI: 10.1016/j.scico.2018.04.002.

[257] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. "Measuring Program Comprehension: A Large-Scale Field Study with Professionals". In: *IEEE Transactions on Software Engineering*. 44. 10. (Oct. 2018). Pp. 951–976. DOI: 10.1109/TSE.2017.2734091.

[258] Jingyi Zhang, Lei Xu, and Yanhui Li. “Classifying Python Code Comments Based on Supervised Learning”. In: *Web Information Systems and Applications*. Cham: Springer International Publishing, 2018. Ed. by Xiaofeng Meng, Ruixuan Li, Kanliang Wang, Baoning Niu, Xin Wang, and Gansen Zhao. Pp. 39–47. DOI: 10.1007/978-3-030-02934-0_4.

[259] Tiago Amorim, Andreas Vogelsang, Florian Pudlitz, Peter Gersing, and Jan Philipps. “Strategies and Best Practices for Model-Based Systems Engineering Adoption in Embedded Systems Industry”. In: *41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019. Pp. 203–212. DOI: 10.1109/icse-seip.2019.00030.

[260] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. “Pareto efficient multi-objective black-box test case selection for simulation-based testing”. In: *Information and Software Technology*. 114. (Oct. 2019). Pp. 137–154. DOI: 10.1016/j.infsof.2019.06.009.

[261] Erki Eessaar and Ege Käosaar. “On Finding Model Smells Based on Code Smells”. In: *Software Engineering and Algorithms in Intelligent Systems*. Cham: Springer International Publishing, May 2019. Ed. by Radek Silhavy. Pp. 269–281. DOI: 10.1007/978-3-319-91186-1_28.

[262] Gidon Ernst, Paolo Arcaini, Alexandre Donze, Georgios Fainekos, Logan Mathesen, Giulia Pedrielli, Shakiba Yaghoubi, Yoriyuki Yamagata, and Zhenya Zhang. “ARCH-COMP 2019 Category Report: Falsification.” In: *ARCH@ CPSIoTWeek*. EasyChair, 2019. Pp. 129–140. DOI: 10.29007/68dk.

[263] Michael Gusenbauer and Neal R. Haddaway. “Which Academic Search Systems are Suitable for Systematic Reviews or Meta-Analyses? Evaluating Retrieval Qualities of Google Scholar, PubMed and 26 other Resources”. In: *Research Synthesis Methods*. 11. 2. (Jan. 2019). Pp. 181–217. DOI: 10.1002/jrsm.1378.

[264] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. “CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines.” In: *Network and Distributed Systems Security (NDSS) Symposium*. Internet Society, 2019. DOI: 10.14722/ndss.2019.23263.

[265] Hao He. “Understanding source code comments at large-scale”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019. Pp. 1217–1219.

[266] Xiao He, Tian Zhang, Minxue Pan, Zhiyi Ma, and Chang-Jun Hu. “Template-based model generation”. In: *Software & Systems Modeling*. 18. 3. (Nov. 2019). Pp. 2051–2092. DOI: 10.1007/s10270-017-0634-5.

[267] Afaq Hussain, Hadeed Ahmed Sher, Ali Faisal Murtaza, and Kamal Al-Haddad. “Improved Restricted Control Set Model Predictive Control (iRCS-MPC) Based Maximum Power Point Tracking of Photovoltaic Module”. In: *IEEE Access*. 7. (2019). Pp. 149422–149432. DOI: 10.1109/ACCESS.2019.2946747.

[268] Anna-Lena Lamprecht, Leyla Garcia, Mateusz Kuzak, Carlos Martinez, Ricardo Arcila, Eva Martin Del Pico, Victoria Dominguez Del Angel, Stephanie van de Sandt, Jon Ison, Paula Andrea Martinez, et al. "Towards FAIR principles for research software". In: *Data Science*. 3. Preprint. (Nov. 2019). Pp. 1–23. DOI: 10.3233/ds-190026.

[269] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. "DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 33. 01. (July 2019). Pp. 1044–1051. DOI: 10.1609/aaai.v33i01.33011044.

[270] Salvador Martínez, Sebastien Gerard, and Jordi Cabot. "On the Need for Intellectual Property Protection in Model-Driven Co-Engineering Processes". In: *Enterprise, Business-Process and Information Systems Modeling*. Cham: Springer International Publishing, 2019. Ed. by Iris Reinhartz-Berger, Jelena Zdravkovic, Jens Gulden, and Rainer Schmidt. Pp. 169–177. DOI: 10.1007/978-3-030-20618-5_12.

[271] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. "Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior". In: *IEEE Transactions on Software Engineering*. 45. 9. (Sept. 2019). Pp. 919–944. DOI: 10.1109/TSE.2018.2811489.

[272] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. "Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior". In: *IEEE Transactions on Software Engineering*. 45. 9. (2019). Pp. 919–944. DOI: 10.1109/TSE.2018.2811489.

[273] Marcus Mikulcak, Paula Herber, Thomas Göthel, and Sabine Glesner. "Information Flow Analysis of Combined Simulink/Stateflow Models". In: *Inf. Technol. Control*. 48. 2. (June 2019). Pp. 299–315. DOI: 10.5755/j01.itc.48.2.21759.

[274] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C. Briand, Stephen Foster, and David Wolfe. "Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn, Estonia: ACM, Aug. 2019. Pp. 1015–1025. DOI: 10.1145/3338906.3340444.

[275] O. Oussalem, M. Kourchi, A. Rachdy, M. Ajaamoum, H. Idadoub, and S. Jenkal. "A low cost controller of PV system based on Arduino board and INC algorithm". In: *Materials Today: Proceedings*. 24. (2019). Pp. 104–109. DOI: 10.1016/j.matpr.2019.07.689.

[276] Vera Pantelic, Alexander Schaap, Alan Wassyng, Victor Bandur, and Mark Lawford. "Something is Rotten in the State of Documenting Simulink Models." In: *MODELSWARD*. SCITEPRESS - Science and Technology Publications, 2019. Pp. 503–510. DOI: 10.5220/0007586005030510.

[277] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. "Categorizing the content of GitHub readme files". In: *Empirical Software Engineering*. 24. 3. (Oct. 2019). Pp. 1296–1327. DOI: 10.1007/s10664-018-9660-3.

[278] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. *Language Models are Unsupervised Multitask Learners*. Tech. rep. OpenAI, 2019. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[279] Dennis Reuling, Udo Kelter, Johannes Bürdek, and Malte Lochau. "Automated N-Way Program Merging for Facilitating Family-Based Analyses of Variant-Rich Software". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 28. 3. (July 2019). 13:1–13:59. DOI: 10.1145/3313789.

[280] Dennis Reuling, Udo Kelter, Sebastian Ruland, and Malte Lochau. "SiMPOSE-Configurable N-Way Program Merging Strategies for Superimposition-Based Analysis of Variant-Rich Software". In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Science, 2019. Pp. 1134–1137. DOI: 10.1109/ase.2019.00120.

[281] Dennis Reuling, Malte Lochau, and Udo Kelter. "From Imprecise N-Way Model Matching to Precise N-Way Model Merging". In: *Journal of Object Technology (JOT)*. 18. 2. (2019). 8:1. DOI: 10.5381/jot.2019.18.2.a8.

[282] Beatriz Sanchez, Athanasios Zolotas, Horacio Hoyos Rodriguez, Dimitris Kolovos, and Richard Paige. "On-the-fly Translation and Execution of OCL-like Queries on Simulink Models". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, Sept. 2019. Pp. 205–215. DOI: 10.1109/models.2019.000-1.

[283] T. Tomita, D. Ishii, T. Murakami, S. Takeuchi, and T. Aoki. "A Scalable Monte-Carlo Test-Case Generation Tool for Large and Complex Simulink Models". In: *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. ACM, May 2019. Pp. 39–46. DOI: 10.1109/MiSE.2019.00014.

[284] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. "A large-scale empirical study on code-comment inconsistencies". In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, May 2019. Pp. 53–64. DOI: 10.1109/icpc.2019.00019.

[285] Oleg A. Yakimenko. *Engineering Computations and Modeling in MATLAB®/Simulink®*. American Institute of Aeronautics and Astronautics, Inc., 2019.

[286] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. "Software documentation: the practitioners' perspective". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, June 2020. Pp. 590–601. DOI: 10.1145/3377811.3380405.

[287] Wesley K. G. Assunção, Silvia R. Vergilio, and Roberto E. Lopez-Herrejon. "Automatic Extraction of Product Line Architecture and Feature Models from UML Class Diagram Variants". In: *Information and Software Technology*. 117. (Jan. 2020). P. 106198. DOI: 10.1016/j.infsof.2019.106198.

[288] Balaji Balasubramaniam, Hamid Bagheri, Sebastian Elbaum, and Justin Bradley. "Investigating Controller Evolution and Divergence through Mining and Mutation". In: *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS).* IEEE, Apr. 2020. Pp. 151–161. DOI: 10.1109/iccps48487.2020.00022.

[289] Sebastian Baltes and Paul Ralph. "Sampling in Software Engineering Research: A Critical Review and Guidelines". In: *Empirical Software Engineering.* 27. 4. (Apr. 2020). DOI: 10.1007/s10664-021-10072-8.

[290] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard, and Claire Pagetti. "CoCoSim, a code generation framework for control/command applications An overview of CoCoSim for multi-periodic discrete Simulink models". In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020).* 2020.

[291] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. "SLEMI: Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink". In: *Proc. 42nd ACM/IEEE International Conference on Software Engineering (ICSE). ACM.* Seoul, South Korea: ACM, June 2020. Pp. 335–346. DOI: 10.1145/3377811.3380381.

[292] Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. "Does your code need comment?" In: *Software: Practice and Experience.* 50. 3. (Nov. 2020). Pp. 227–245. DOI: 10.1002/spe.2772.

[293] M. Jaskolka, V. Pantelic, A. Wassyng, and M. Lawford. "A Comparison of Componentization Constructs for Supporting Modularity in Simulink". In: *SAE Technical Paper.* 1. 2020-01-1290. (Apr. 2020). DOI: 10.4271/2020-01-1290.

[294] Monika Jaskolka, Vera Pantelic, Alan Wassyng, and Mark Lawford. *Supporting Modularity in Simulink Models.* 2020. DOI: 10.48550/arXiv.2007.10120.

[295] Monika Jaskolka, Stephen Scott, Vera Pantelic, Alan Wassyng, and Mark Lawford. "Applying modular decomposition in Simulink". In: *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'20).* IEEE, Oct. 2020. Pp. 31–36. DOI: 10.1109/issrew51248.2020.00033.

[296] MathWorks Advisory Board. *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow – Version 5.* Tech. rep. The MathWorks, Inc, 2020, p. 272. URL: https://de.mathworks.com/content/dam/mathworks/mathworks-dot-com/solutions/mab/mab-control-algorithm-modeling-guidelines-using-matlab-simulink-and-stateflow-v5.pdf (visited on 10/27/2025).

[297] Daniel Mendez, Daniel Graziotin, Stefan Wagner, and Heidi Seibold. "Open Science in Software Engineering". In: *Contemporary Empirical Methods in Software Engineering.* Ed. by Michael Felderer and Guilherme Horta Travassos. Cham: Springer International Publishing, 2020, pp. 477–501. ISBN: 9783030324896. DOI: 10.1007/978-3-030-32489-6_17. URL: https://doi.org/10.1007/978-3-030-32489-6_17.

[298] Vishal Misra, Jakku Sai Krupa Reddy, and Sridhar Chimalakonda. "Is there a correlation between code comments and issues? an exploratory study". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing.* 2020. Pp. 110–117.

[299] Peter Munk and Arne Nordmann. "Model-based safety assessment with SysML and component fault trees: application and lessons learned". In: *Software and Systems Modeling*. 19. 4. (Feb. 2020). Pp. 889–910. DOI: 10.1007/s10270-020-00782-w.

[300] Nebras Nassar, Jens Kosiol, Timo Kehrer, and Gabriele Taentzer. "Generating Large EMF Models Efficiently - A Rule-Based, Configurable Approach". In: *Fundamental Approaches to Software Engineering*. Springer, 2020. Pp. 224–244. DOI: 10.1007/978-3-030-45234-6_11.

[301] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. "MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools". In: *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, Dec. 2020. Pp. 1103–1115. DOI: 10.1145/3324884.3416668.

[302] Paul Ralph, Sebastian Baltes, Domenico Bianculli, Yvonne Dittrich, Michael Felderer, and *et al. ACM SIGSOFT Empirical Standards*. 2020. DOI: 10.48550/arXiv.2010.03525.

[303] Sayed Mohsin Reza, Omar Badreddin, and Khandoker Rahad. "ModelMine: a tool to facilitate mining models from open source repositories". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. Virtual Event, Canada: ACM, Oct. 2020. 9:1–9:5. DOI: 10.1145/3417990.3422006.

[304] Alexander Schlie, Alexander Knüppel, Christoph Seidl, and Ina Schaefer. "Incremental feature model synthesis for clone-and-own software systems in MATLAB/Simulink". In: *Proceedings of the 24th ACM Conference on Systems and Software Product Lines*. ACM, Oct. 2020. Pp. 1–12. DOI: 10.1145/3382025.3414973.

[305] Alexander Schlie, Sandro Schulze, and Ina Schaefer. "Recovering variability information from source code of clone-and-own software systems". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. Magdeburg, Germany: ACM, Feb. 2020. Pp. 1–9. DOI: 10.1145/3377024.3377034.

[306] Alexander Schultheiß, Paul Maximilian Bittner, Timo Kehrer, and Thomas Thüm. "On the use of product-line variants as experimental subjects for clone-and-own research: a case study". In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A*. ACM, Oct. 2020. Pp. 1–6. DOI: 10.1145/3382025.3414972.

[307] Sohil Lal Shrestha. "Automatic generation of Simulink models to find bugs in a cyber-physical system tool chain using deep learning". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ACM, June 2020. Pp. 110–112. DOI: 10.1145/3377812.3382163.

[308] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. "DeepFuzzSL: Generating models with deep learning to find bugs in the Simulink toolchain". In: *2nd workshop on testing for deep learning and deep learning for testing (DeepTest)*. ACM, May 2020.

[309] Andreas Vogelsang. “Feature dependencies in automotive software systems: Extent, awareness, and refactoring”. In: *Journal of Systems and Software*. 160. (Feb. 2020). Pp. 1–15. DOI: 10.1016/j.jss.2019.110458.

[310] Andreas Vogelsang, Jonas Eckhardt, Daniel Mendez, and Moritz Berger. “Views on quality requirements in academia and practice: commonalities, differences, and context-dependent grey areas”. In: *Information and Software Technology*. 121. (May 2020). P. 106253. DOI: 10.1016/j.infsof.2019.106253.

[311] Arianna Blasi, Nataliia Stulova, Alessandra Gorla, and Oscar Nierstrasz. “RepliComment: Identifying clones in code comments”. In: *Journal of Systems and Software*. 182. (Dec. 2021). P. 111069. DOI: 10.1016/j.jss.2021.111069.

[312] Virginia Braun and Victoria Clarke. *Thematic Analysis: A Practical Guide*. SAGE Publications Ltd, Dec. 2021, p. 376.

[313] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. “Sampling Projects in GitHub for MSR Studies”. In: *MSR*. IEEE, May 2021. Pp. 560–564. DOI: 10.1109/msr52588.2021.00074.

[314] Monika Jaskolka, Vera Pantelic, Alan Wassyng, Mark Lawford, and Richard Paige. “Repository Mining for Changes in Simulink Models”. In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, Oct. 2021. Pp. 46–57. DOI: 10.1109/MODELS50736.2021.00014.

[315] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. “History-Based Model Repair Recommendations”. In: *ACM Trans. Softw. Eng. Methodol.* 30. 2. (Jan. 2021). Pp. 1–46. DOI: 10.1145/3419017.

[316] Saheed Popoola and Jeff Gray. “Artifact Analysis of Smell Evolution and Maintenance Tasks in Simulink Models”. In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, Oct. 2021. Pp. 817–826. DOI: 10.1109/models-c53483.2021.00128.

[317] Pooja Rani, Suada Abukar, Nataliia Stulova, Alexandre Bergel, and Oscar Nierstrasz. “Do comments follow commenting conventions? a case study in Java and Python”. In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Morressier, Sept. 2021. Pp. 165–169. DOI: 10.26226/morressier.613b54401459512fce6a7d00.

[318] Pooja Rani, Mathias Birrer, Sebastiano Panichella, Mohammad Ghafari, and Oscar Nierstrasz. “What do developers discuss about code comments?” In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Morressier, Sept. 2021. Pp. 153–164. DOI: 10.26226/morressier.613b54401459512fce6a7cff.

[319] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. “How to identify class comment types? A multi-language approach for class comment classification”. In: *Journal of Systems and Software*. 181. (Nov. 2021). P. 111047. DOI: 10.1016/j.jss.2021.111047.

[320] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Mohammad Ghafari, and Oscar Nierstrasz. "What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk". In: *Empirical Software Engineering*. 26. 6. (Aug. 2021). P. 112. DOI: 10.1007/s10664-021-09981-5.

[321] Simone Romano, Maria Caulo, Matteo Buompastore, Leonardo Guerra, Anas Mounsif, Michele Telesca, Maria Teresa Baldassarre, and Giuseppe Scanniello. "G-Repo: A Tool to Support MSR Studies on GitHub". In: *SANER*. IEEE, Mar. 2021. Pp. 551–555. DOI: 10.1109/saner50967.2021.00064.

[322] Andreas Schlie. "Extractive Product Line Migration for MATLAB/Simulink Software Systems". PhD thesis. Technische Universität Carolo-Wilhelmina zu Braunschweig, 2021.

[323] Alexander Schultheiß, Paul Maximilian Bittner, Lars Grunske, Thomas Thüm, and Timo Kehrer. "Scalable N-Way Model Matching Using Multi-Dimensional Search Trees". In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2021. Pp. 1–12. DOI: 10.1109/MODELS50736.2021.00010.

[324] Alexander Schultheiß, Paul Maximilian Bittner, Lars Grunske, Thomas Thüm, and Timo Kehrer. "Scalable N-Way Model Matching Using Multi-Dimensional Search Trees". In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE Computer Science, Oct. 2021. Pp. 1–12. DOI: 10.1109/models50736.2021.00010.

[325] Oszkár Semeráth, Aren A. Babikian, Boqi Chen, Chuning Li, Kristóf Marussy, Gábor Szárnyas, and Dániel Varró. "Automated generation of consistent, diverse and structurally realistic graph models". In: *Software and Systems Modeling*. 20. 5. (May 2021). Pp. 1713–1734. DOI: 10.1007/s10270-021-00884-z.

[326] Sohil Lal Shrestha and Christoph Csallner. "SLGPT: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain". In: *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. Trondheim, Norway: ACM, June 2021. Pp. 260–265. DOI: 10.1145/3463274.3463806.

[327] Alexander Boll, Nicole Vieregg, and Timo Kehrer. *Replicability of Experimental Tool Evaluations in Model-Based Software and Systems Engineering with MATLAB/Simulink*. 2022. DOI: 10.6084/m9.figshare.13633928.

[328] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. "Low-code development and model-driven engineering: Two sides of the same coin?" en. In: *Software and Systems Modeling*. 21. 2. (Apr. 2022). Pp. 437–446. DOI: 10.1007/s10270-021-00970-2.

[329] Michael Goedicke and Ulrike Lucke. "Research Data Management in Computer Science - NFDIxCS Approach". In: *52. Jahrestagung der Gesellschaft für Informatik, INFORMATIK 2022, Informatik in den Naturwissenschaften, 26. - 30. September 2022, Hamburg*. Bonn, Germany: Gesellschaft für Informatik, Bonn, 2022. Ed. by Daniel Demmler, Daniel Krupka, and Hannes Federrath. Pp. 1317–1328.

[330] Shikai Guo, He Jiang, Zhihao Xu, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Rong Chen. "Detecting Simulink compiler bugs via controllable zombie blocks mutation". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore, Singapore: ACM, Nov. 2022. Pp. 1061–1072. DOI: 10.1145/3540250.3549159.

[331] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. 4th ed. Elsevier, 2022. ISBN: 9780123814791.

[332] Marija Kostić, Aleksa Srbljanović, Vuk Batanović, and Boško Nikolić. "Code comment classification taxonomies". In: *Proceedings of the Ninth IcETRAN Conference*. 2022.

[333] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. "Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach". In: *IEEE Transactions on Software Engineering*. 49. 4. (Apr. 2022). Pp. 1640–1660. DOI: 10.1109/tse.2022.3185458.

[334] José Antonio Hernández López and Jesús Sánchez Cuadrado. "An efficient and scalable search engine for models". In: *Software and Systems Modeling*. 21. 5. (Dec. 2022). Pp. 1715–1737. DOI: 10.1007/s10270-021-00960-4.

[335] National Information Standards Organization (NISO). *ANSI/NISO Z39.104-2022, CRediT, Contributor Roles Taxonomy*. Feb. 8, 2022. DOI: 10.3789/ansi.niso.z39.104-2022. URL: https://www.niso.org/publications/z39104-2022-credit.

[336] Alexander Schultheiss et al. *Artifact for RaQuN: A Generic and Scalable N-Way Model Matching Algorithm*. Version SoSyM-Submission. Software artifact including source code and Docker image for RaQuN. Related GitHub repository: https://github.com/AlexanderSchultheiss/RaQuN/tree/SoSyM-Submission. Mar. 2022. DOI: 10.5281/zenodo.7372740.

[337] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. "SLNET: A Redistributable Corpus of 3rd-party Simulink Models". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. Pittsburgh, Pennsylvania: ACM, May 2022. Pp. 237–241. DOI: 10.1145/3524842.3528001.

[338] SPES_XT Consortium. *Software Platform Embedded Systems 2020 (SPES_XT)*. Project website. 2022. URL: https://sse.uni-due.de/en/research/projects/spes-2020 (visited on 10/29/2025).

[339] Murali Sridharan, Mika Mäntylä, Maëlick Claes, and Leevi Rantala. "SoCCMiner: a source code-comments and comment-context miner". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022. Pp. 242–246.

[340] Christopher Vendome, Eric J. Rapos, and Nick DiGennaro. “How do I model my system? A Qualitative Study on the Challenges that Modelers Experience”. In: *ICPC*. ACM, May 2022. Pp. 648–659. DOI: 10.1145/3524610.3529160.

[341] Xiaowei Zhang, Weiqin Zou, Lin Chen, Yanhui Li, and Yuming Zhou. “Towards the Analysis and Completion of Syntactic Structure Ellipsis for Inline Comments”. In: *IEEE Transactions on Software Engineering*. 49. 4. (Apr. 2022). Pp. 2285–2302. DOI: 10.1109/tse.2022.3216279.

[342] Alexander Boll, Tiago Amorim, Ferry Bachmann, Timo Kehrer, Andreas Vogelsang, and Hartmut Pohlheim. *SimuComp Bus Study Data Set*. 2023. DOI: 10.5281/zenodo.8011478.

[343] Yuan Huang, Hanyang Guo, Xi Ding, Junhuai Shu, Xiangping Chen, Xiapu Luo, Zibin Zheng, and Xiaocong Zhou. “A comparative study on method comment and inline comment”. In: *ACM Transactions on Software Engineering and Methodology*. 32. 32. (2023). Pp. 1–26.

[344] Marcel Jerzyk and Lech Madeyski. “Code Smells: A Comprehensive Online Catalog and Taxonomy”. In: *Developments in Information and Knowledge Management Systems for Business Applications: Volume 7*. Springer, 2023, pp. 543–576.

[345] Marija Kostić, Vuk Batanović, and Boško Nikolić. “Monolingual, multilingual and cross-lingual code comment classification”. In: *Engineering Applications of Artificial Intelligence*. 124. (2023). Pp. 1–17.

[346] Firas Al Laban, Jan Bernoth, Michael Goedicke, Ulrike Lucke, Michael Striewe, Philipp Wieder, and Ramin Yahyapour. “Establishing the Research Data Management Container in NFDIxCS”. In: *1st Conference on Research Data Infrastructure - Connecting Communities, CoRDI 2023, Karlsruhe, Germany, September 12-14, 2023*. New York City, NY, USA: TIB Open Publishing, Sept. 2023. Ed. by York Sure-Vetter and Carole A. Goble. DOI: 10.52825/cordi.v1i.395.

[347] Licensee. *licensee*. 2023. URL: https://github.com/licensee/licensee (visited on 10/27/2025).

[348] Shifan Liu, Zhanqi Cui, Xiang Chen, Jun Yang, Li Li, and Liwei Zheng. “TBCUP: A Transformer-based Code Comments Updating Approach”. In: *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, June 2023. Pp. 892–897. DOI: 10.1109/compsac57700.2023.00119.

[349] Haoyang Ma. *A Survey of Modern Compiler Fuzzing*. 2023. DOI: 10.48550/arXiv.2306.06884.

[350] MES. *Modeling Guidelines Interest Group (MGIGroup)*. Link was active as of: 06/06/2023. 2023. URL: https://model-engineers.com/en/academy/mgigroup.

[351] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. “A decade of code comment quality assessment: A systematic literature review”. In: *Journal of Systems and Software*. 195. (Jan. 2023). P. 111515. DOI: 10.1016/j.jss.2022.111515.

[352] Sohil Lal Shrestha. “Constructing Large Open-Source Corpora and Leveraging Language Models for Simulink Toolchain Testing and Analysis”. Dissertation. The University of Texas at Arlington, 2023.

[353] Sohil Lal Shrestha. “Harnessing Large Language Models for Simulink Toolchain Testing and Developing Diverse Open-Source Corpora of Simulink Models for Metric and Evolution Analysis”. In: *ISSTA*. ACM, July 2023. Pp. 1541–1545. DOI: 10.1145/3597926.3605233.

[354] Sohil Lal Shrestha, Alexander Boll, Timo Kehrer, and Christoph Csallner. *ScoutSL Artifacts*. Aug. 2023. DOI: 10.6084/m9.figshare.23717763.v1.

[355] Sohil Lal Shrestha, Alexander Boll, Timo Kehrer, and Christoph Csallner. *ScoutSL: ScoutSL Simulink Search Engine*. Version v1.2. Aug. 2023. DOI: 10.5281/zenodo.8266234.

[356] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. “Replicability Study: Corpora For Understanding Simulink Models & Projects”. In: *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Oct. 2023. Pp. 1–12. DOI: 10.1109/esem56168.2023.10304867.

[357] Florian Sihler, Matthias Tichy, and Jakob Pietron. “One-Way Model Transformations in the Context of the Technology-Roadmapping Tool IRIS”. In: *Journal of Object Technology*. 22. 2. (July 2023). 2:1–14. DOI: 10.5381/jot.2023.22.2.a2.

[358] T. Dohmke. *100 million developers and counting*. 2023. URL: https://github.blog/2023-01-25-100-million-developers-and-counting/ (visited on 10/27/2025).

[359] Juho Tevajärvi. “Protecting Intellectual Property in Multi-Supplier Ship Powertrain Co-Simulation”. Master’s Thesis. Otaniemi: Aalto University, Dec. 2023.

[360] Chao Wang, Hao He, Uma Pal, Darko Marinov, and Minghui Zhou. “Suboptimal comments in Java projects: From independent comment changes to commenting practices”. In: *ACM Transactions on Software Engineering and Methodology*. 32. 2. (Mar. 2023). Pp. 1–33. DOI: 10.1145/3546949.

[361] Mingyi Zhou, Xiang Gao, Jing Wu, John Grundy, Xiao Chen, Chunyang Chen, and Li Li. “ModelObfuscator: Obfuscating Model Information to Protect Deployed ML-Based Systems”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, July 13, 2023. Pp. 1005–1017. DOI: 10.1145/3597926.3598113.

[362] Bhisma Adhikari, Eric J. Rapos, and Matthew Stephan. “SimIMA: a virtual Simulink intelligent modeling assistant”. en. In: *Software and Systems Modeling*. 23. 1. (Feb. 2024). Pp. 29–56. DOI: 10.1007/s10270-023-01093-6.

[363] Arsene Indamutsa, Juri Di Rocco, Lissette Almonte, Davide Di Ruscio, and Alfonso Pierantonio. “Advanced discovery mechanisms in model repositories”. In: *Software: Practice and Experience*. 54. 11. (2024). Pp. 2214–2248.

[364] Xiaochen Li, Shikai Guo, Hongyi Cheng, and He Jiang. “Simulink Compiler Testing via Configuration Diversification With Reinforcement Learning”. In: *IEEE Transactions on Reliability*. 73. 2. (June 2024). Pp. 1060–1074. DOI: 10.1109/TR.2023.3317643.

[365] Christof Tinnes, Alisa Welter, and Sven Apel. *Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets.* Apr. 2024. DOI: 10.1109/icse55347.2025.00112.

[366] Thomas Weber and Sebastian Weber. “Model Everything but with Intellectual Property Protection - The Deltachain Approach”. In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems.* Linz, Austria: ACM, Sept. 2024. Pp. 49–56. DOI: 10.1145/3640310.3674086.

[367] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. “Fuzz4All: Universal Fuzzing with Large Language Models”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* Lisbon, Portugal: ACM, Apr. 2024. Pp. 1–13. DOI: 10.1145/3597503.3639121.

[368] Linghan Huang, Peizhou Zhao, Huaming Chen, and Lei Ma. *On the Challenges of Fuzzing Techniques via Large Language Models.* July 2025. DOI: 10.1109/sse67621.2025.00028.

[369] Yilin Huang. “Reproducibility and Replicability of Simulation Models”. In: *2025 Annual Modeling and Simulation Conference (ANNSIM).* 2025. Pp. 1–10.

[370] Michał Markiewicz and Lesław Gniewek. “Review of hierarchy in Petri Nets”. In: *International Journal of Electronics and Telecommunication.* 71. 3. (July 2025). Pp. 1–9. DOI: 10.24425/ijet.2025.155452.

[371] Pooja Rani, Jan-Andrea Bard, June Sallou, Alexander Boll, Timo Kehrer, and Alberto Bacchelli. *Can We Make Code Green? Understanding Trade-Offs in LLMs vs. Human Code Optimizations.* 2025. URL: https://arxiv.org/abs/2503.20126.

[372] Zehong Yu, Yixiao Yang, Zhuo Su, Rui Wang, Yang Tao, and Yu Jiang. “Knight: Optimizing Code Generation for Simulink Models With Loop Reshaping”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* 44. 2. (Feb. 2025). Pp. 444–457. DOI: 10.1109/TCAD.2024.3438691.

[373] Jingfan Zhang, Delaram Ghobari, Mehrdad Sabetzadeh, and Shiva Nejati. “Simulink Mutation Testing using CodeBERT”. In: *2025 IEEE/ACM International Conference on Automation of Software Test (AST).* IEEE, Apr. 2025. Pp. 24–28. DOI: 10.1109/AST66626.2025.00009.

[374] *Case Study on Structural Views for Component and Connector Models Website.* URL: https://www.se-rwth.de/materials/cncviewscasestudy/ (visited on 10/29/2025).

[375] CREST Consortium. *Collaborative Embedded Systems (CREST).* URL: https://crest.in.tum.de/ (visited on 10/27/2025).

[376] ETAS. *ASCET-DEVELOPER – Model-based design and auto c-code generation for embedded systems.* URL: https://www.etas.com/en/products/ascet-developer.php (visited on 10/27/2025).

[377] GitHub Inc. *About.* URL: https://github.com/about (visited on 10/27/2025).

[378] Google. *BigQuery.* URL: https://cloud.google.com/bigquery (visited on 10/27/2025).

[379] Misra. *MISRA AC SLSF*. Tech. rep. MISRA. URL: https://www.misra.org.uk/product/misra-ac-slsf/ (visited on 10/27/2025).

Erklärung

gemäß Art. 18 PromR Phil.-nat. 2019

Name/Vorname:	Boll, Alexander
Matrikelnummer:	21-141-585
Studiengang:	PhD of Science in Computer Science
	Bachelor ☐ Master ☐ Dissertation [✓ ☒]
Titel der Arbeit:	**Bridging the Data Desert: Mitigating Challenges of Model Accessibility in Simulink Research**
Leiter der Arbeit:	Prof. Dr. Timo Kehrer Universität Bern

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäß Artikel 36 Absatz 1 Buchstabe r des Gesetzes über die Universität vom 5. September 1996 & Artikel 69 des Universitätsstatuts vom 7. Juni 2011 zum Entzug des Doktortitels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbstständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die Doktorarbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 31. Oktober 2025	[illegible]
Ort/Datum	Unterschrift