Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts

Thomas Gerlitz

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2017-8
RWTH Aachen University | Department of Computer Science | September 2017

Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Ingenieurswissenschaften genehmigte Dissertation

vorgelegt von

Thomas Gerlitz, M.Sc. RWTH

aus Neuss

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr.-Ing. Ina Schaefer

Tag der mündlichen Prüfung: 29. Mai 2017

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Thomas Gerlitz
Lehrstuhl Informatik 11 - Embedded Software
gerlitz@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2017-08

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

In recent years, the automotive industry adopted model-based development of software components as an integral part of the automotive software development process. The use of models enables the simulation and testing of system behavior even in early stages of development. They can further serve as input to code generators, allowing the model-based development of firmware for complex systems such as the electronic control units (ECU) of an automobile. As the complexity and size of models continues to grow, a need has arisen for dynamic and static model analysis techniques to keep costs for quality assessment as required by norms and standards such as ISO 26262 in check. While a plethora of tools exists for the analysis of software source-code, the tool landscape for the analysis of software models is still sparsely populated. Due to abstractions used within models and their heterogeneous and often proprietary file formats, the realization of generic model analysis tools cannot be performed to the same extent as for software source-code.

In this thesis, we present a method for the incremental integration and static analysis of model-based software artifacts comprising the extraction, storage, analysis and evolution of model data. The proposed incremental integration approach allows the conversion of supported artifacts into a well-defined representation and subsequent storage in a model repository, enabling seamless access to stored artifacts as well as synchronization with changes made to their source models. We further propose multiple static analysis techniques for MATLAB/Simulink models, a prevalent model-based software artifact in automotive software development. These analyses support various activities during different stages of a model-based development process. We present a signal reconstruction and slicing algorithm that supports debugging, testing and exploration activities of MATLAB/Simulink models. A clone detection procedure allows the automatic identification of cloned model fragments and their subsequent controlled reuse by refactoring into generic library blocks. Further quality and design defects are detected by a model smell analysis, identifying anti-patterns that negatively influence quality properties of MATLAB/Simulink models. Furthermore, we propose an inter-artifact consistency analysis targeting traceability links between artifacts of a product line and its accompanying variability documentation. All proposed techniques are realized in the form of an integrated software framework called artshop.

To show the applicability of the artshop framework, we applied the realized techniques on a set of real-world models taken from academic and industrial case studies to assess the overall scalability and performance of the framework. We show that the integration and analysis capabilities of the artshop framework are already applicable to real-world models.

Zusammenfassung

In den letzten Jahren setzt die Automobilindustrie für die Entwicklung von Softwarekomponenten verstärkt auf die modellbasierte Softwareentwicklung. Die Verwendung von Modellen ermöglicht bereits in frühen Entwicklungsstadien die Simulation und das Testen von Verhaltensmodellen sowie die Generation des Programmcodes, welcher schlussendlich auf die Steuergeräte eines Automobils geladen wird. Mit steigender Größe und Komplexität verwendeter Modelle gewinnen Qualitätsanalysen in Form von statischen und dynamischen Modelanalysen an Bedeutung, um die Gesamtkosten der Qualitätssicherung, wie sie von Normen und Standards wie der ISO-26262 gefordert wird, zu reduzieren. Während auf dem Markt eine Vielzahl von Analysewerkzeugen für Software-Quellcode zur Verfügung stehen, existieren nur wenige Werkzeuge, welche Qualitätsanalysen direkt auf Modellebene umsetzen. Durch die in Modellen üblichen Abstraktionen und ihre oftmals heterogenen proprietären Dateiformate ist die Realisierung von generischen Modellanalysewerkzeugen nicht mit ähnlichem Aufwand und Vorgehen realisierbar wie für Software-Quellcode.

In dieser Arbeit präsentieren wir einen Ansatz für die inkrementelle Integration und statische Analyse von modellbasierten Softwareartefakten, welcher die Extraktion, Verwaltung, Evolution und Analyse von Modelldaten umfasst. Der vorgestellte Ansatz zur inkrementellen Integration erlaubt die Extraktion und Konvertierung von unterstützten Artefakten in eine definierte Modellrepräsentation und Ablage in einem Modellrepository. Dieses ermöglicht den nahtlosen Zugriff auf die darin gespeicherten Artefakte und die Synchronisation von Änderungen, die an den Quellartefakten außerhalb des Repositories vorgenommen wurden. Eines der weitverbreitetsten Artefakte in der Automobilindustrie sind MATLAB/Simulink Modelle. In dieser Arbeit werden statische Analysetechniken für diese Art von Modellen diskutiert, welche Aktivitäten innerhalb eines modellbasierten Entwicklungsprozesses unterstützen. Unter anderem stellen wir einen Signalrekonstruktions- und Slicingalgorithmus vor, welcher das Debugging, Testen und die Exploration von MATLAB/Simulink Modellen unterstützt. Ein Klonerkennungsprozess erleichtert die Detektion von duplizierten Modellfragmenten und ermöglicht deren kontrollierte Wiederverwendung durch Modelltransformationen, welche gefundene Klone mit Hilfe eines generischen, parametrierbaren Bibliotheksblocks refaktorieren können. Weitere Qualitäts- und Designdefizite können durch eine Model-Smell-Analyse detektiert werden, welche Anti-Pattern in MATLAB/Simulink Modellen identifiziert die Qualitätseigenschaften eines Modells negativ beeinflussen. Weiterhin führen wir eine artefaktübergreifende Konsistenzprüfung von Traceability-Links zwischen Artefakten einer Produktlinie und deren Variabilitätsdokumentation ein. Die in dieser Arbeit diskutierten Techniken wurden im experimentellen Softwareframework artshop implementiert.

Um die praktische Anwendbarkeit von den in artshop realisierten Techniken zu demonstrieren, wurden diese auf eine Menge von Modellen aus akademischen und industriellen Fallstudien angewendet um deren Skalierbarkeit und Performanz auf realen Beispielen zu beurteilen. Dabei kommen wir zu dem Schluss, dass die Integrations- und Analysemöglichkeiten von artshop bereits auf reale Modelle anwendbar sind.

Acknowledgments

This thesis was created as part of my activities as a research assistant at the chair of embedded software at RWTH Aachen University. The creation of this thesis would not have been possible without the support of many others that accompanied me during my time as a research assistant.

First, I would like to thank Prof. Dr.-Ing. Stefan Kowalewski for giving me the opportunity to join his group, supporting my thesis and for the great collaboration during this time. I would also like to thank Prof. Dr.-Ing. Ina Schaefer for serving as the second supervisor of this thesis. Furthermore, I thank Prof. Dr. Stefan Decker and Prof. Dr. Jürgen Giesl for participating in my examination committee.

I have to thank Christian Dernehl and Norman Hansen who were always open for discussions about various concepts of MATLAB/Simulink and animated me to drive the capabilities of the artshop framework further. I would also like to thank Quang Minh-Tran from the DCAITI for fruitful discussions and collaborations on analyses and model transformations on MATLAB/Simulink models.

Special thanks go to all my former colleagues and friends for constructive discussions (un)related to my area of research, welcome work environment and the awesome time I had at the chair during both research and non-research related activities.

Moreover, I would like to thank all my students for their active support during the development of the techniques described in this thesis. This thesis would not have been possible without the contributions to algorithms, user-interfaces, test cases and the maintenance of the projects infrastructure by my students. In particular, Stefan Schake and Mirko Kugelmeier have contributed a lot of effort towards the successful realization of the techniques within this thesis.

I also thank everyone that reviewed my thesis for grammatical and spelling errors and thereby contributed to the successful completion of this thesis.

Finally, I would like to thank my family for the backing during my studies and Jennifer Janas for the never-ending support especially during the final stages of this thesis.

Thomas Gerlitz
February 2017, Aachen

Contents

List of Tables

List of Figures

1 Motivation

In the past decade, the automotive industry adopted and backed model-based development of software components in an effort to yield a substantial increase in productivity and software quality. The abstraction capabilities enabled by structural and behavioral models allow the use of traditional mechanical engineering methods and domain specific concepts such as function block diagrams for modeling control algorithms instead of writing code. Code generators then allow engineers without prior software development experience to turn modeled diagrams into optimized processor firmware. It is further possible to simulate and verify models even in the early stages of development. One of the most established tools for model-based development in the automotive domain is MATLAB/Simulink developed by The Mathworks [46]. It allows the modeling of dynamic systems and control algorithms using functional block diagrams and further includes a code generator.

With the increased use of model-based software development in the automotive domain, models slowly replace source code as the main development artifact. Since the source code of the software deployed on the individual ECUs in an automobile is generated from models, the quality of the model directly influences the quality of the generated software [53]. Quality control of automotive software is fundamental to the automotive industry, as errors in the software cost billions due to recalls and warranties and may further endanger the lives of passengers or other road users [1]. Since 90 % of innovations made in the automotive domain are driven by software and electronics [18], quality assurance is a necessity during the development process and required by safety norms such as the ISO 26262 [73]. While there exists a huge range of syntactic, semantic and structural software source code analysis tools, the tool landscape for tools analyzing the quality of software models is still sparse. One reason is that the primary representation for source code is text observing a well-defined syntax. Models on the other hand are typically saved in proprietary file formats with no explicit documentation of the model files structure, imposing a higher effort to extract relevant model data and processing it in a format suitable for quality analyses. Applying source code quality analysis to code generated from model files would require more time and effort, as source code is usually not generated until late development stages and mapping found flaws to constructs within the respective models might prove to be infeasible in certain cases. Nevertheless, with rising complexity and size of models, quality analysis of models in early development stages is important for cost-efficient removal of quality defects. Hence, quality analyses are needed that directly target the actual model-based software artifacts to increase the overall quality of models starting from early development stages.

1.1 Problem and Objectives

In this thesis, we address multiple problems arising during the integration and analysis of artifacts of a typical model-based software development process in the automotive domain. As models are saved in heterogeneous proprietary file formats, uniform and structural storage of model files is not possible. While certain artifacts are saved as files, others are stored in databases of their respective development tools and are not locally available on the client system. Thus, tracing changes on a structural level is only possible if directly supported by the individual development tool as the flat textual representation cannot be mapped to the often-complex structures contained in a software model. As models grow in size and complexity throughout the development process, in particular during the development of MATLAB/Simulink models, manual quality assessment of models containing tens of thousands of model elements becomes impractical.

Hence, prior to the development of quality analyses in the form of static model analyses, the accessibility of models and their encapsulated information must be ensured by extracting and integrating artifact data in well-defined data structures. A well-defined interface to models and their data structures enables seamless processing of model data and the development of static model analysis techniques that target quality properties of software models or support a developer during common development tasks such as debugging, testing or change-impact analysis. These techniques can then be applied for early-stage quality assessment, reducing the overall cost of the development process. As MATLAB/Simulink models are one of the primarily used models within the automotive domain, analyses targeting these models are of particular importance.

1.2 Solution Approach

The aim of this thesis is to provide a framework for the integration and static analysis of model-based software artifacts comprising the extraction, storage, analysis and evolution of model data as shown in the process depicted in Figure 1.1.

Artifact Integration In the first phase of this process, model data shall be extracted from their respective sources, converted into a well-defined representation and integrated into a model repository supporting version control of stored data on a structural level. To support the model conversion process, a metamodel shall be defined which serves as a foundation for the representation of extracted model data. The metamodel shall also contain structures expressing traceability links between arbitrary model elements, enabling inter-artifact consistency analyses.

Model Analysis Artifact data in the model repository shall be available for the conduction of model analyses. Thus, all kinds of static model analysis techniques for the analysis of single or set of artifacts can be realized within the framework.

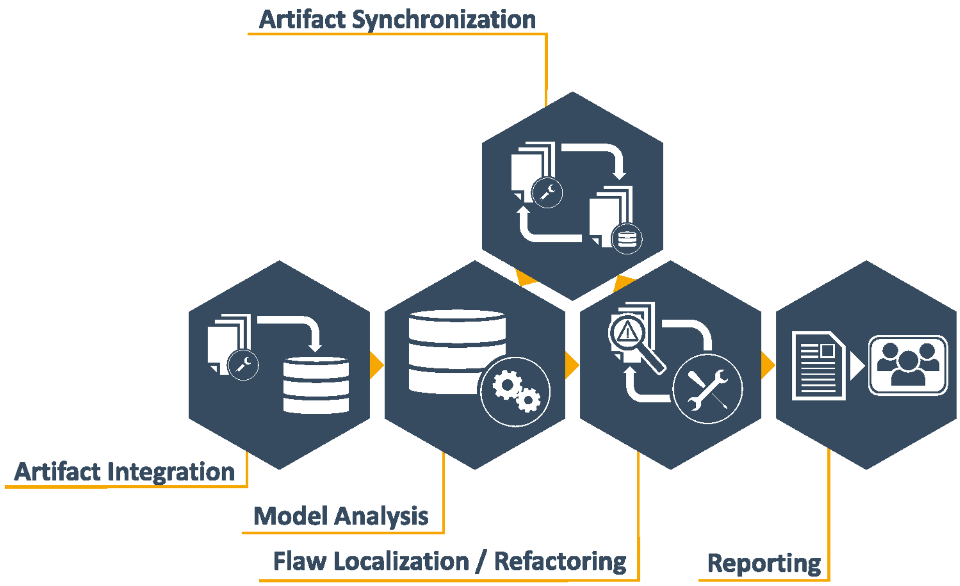


Figure 1.1: Exemplary model analysis process

Flaw Localization / Refactoring Results of analyses must be traceable to their respective occurrence. As development tools such as MATLAB/Simulink only have limited visualization capabilities, graphical views shall be included in the framework to realize arbitrary visualization of analysis findings. To further ease the resolution of found defects, refactoring operations shall be included to assist a developer during the resolution of defects.

Artifact Synchronization To continuously monitor the state of an artifact during its evolution as part of the development process, the framework shall support the synchronization of already imported artifacts with their source artifacts in an incremental fashion, i.e. the change Δ between the imported and current state of the source artifact is identified and applied to the version stored in the repository.

Reporting A common tool for the communication of defects to the various stakeholder of an artifact are reports. These summarize the findings of analyses and document the state of one or multiple artifacts with regard to the analyzed model characteristics.

1.3 Contributions

The main contributions of this thesis are:

- An approach for the **incremental integration** of model-based software artifacts. By supplying a tool-independent metamodel, **tool adapters** define specializations of this metamodel to create model representations of tool specific artifacts. Three adapters are defined to extract model data from the tools MATLAB/Simulink, IBM Rational DOORS and pure::variants and provide **customizable views** on these artifacts. The data extracted by these adapters can be saved within a **model repository** for the subsequent use in model analyses. Additionally, a **synchronization mechanism** is proposed that is able to identify changes between an imported representation and its source artifact and apply them to the representation stored within the repository, while simultaneously preserving the history of the synchronized artifact.

- A method for the **extraction of feature mappings** from the variability documentation of a product line and a subsequent **feature consistency analysis** to check the correctness of the extracted mapping. This mapping can be stored within the model repository and be used for further consistency or feature-based analyses of the artifacts of a product line.

- A **signal reconstruction**, **dependency analysis** and **flow-sensitive slicing approach** for MATLAB/Simulink models. Signal relationships within a MATLAB/Simulink model are complex and partially obscured through the application of architectural patterns within a model. The reconstruction of all atomic and composite signals in a model allows the creation of comprehensive analyses on the signals of a MATLAB/Simulink model. We use the signal information for the computation of static forward and backward slices within a MATLAB/Simulink model with respect to a certain point of interest.

- A **clone detection** and **refactoring** procedure for MATLAB/Simulink models. This approach uses two known detectors from the literature in addition to a novel **layout-based copy-clone detector**. By **consolidating** the result set of all three detectors, i.e. removal of duplicate occurrences and subsuming clones representing subsets of other clones, the overall quality of the result set of individual clone detectors can be increased. Furthermore, a **refactoring procedure** for detected clones is defined, which eases the controlled reuse of clones.

- The **definition** of a set of **model smells** for MATLAB/Simulink models, i.e. anti-pattern, which negatively affect the overall quality of a model but do not necessarily represent semantic or syntactic errors. Model smells are defined for a number of different categories. We further created a **detector** for these smells, based on a validation language that can directly be specified within the proposed framework, allowing a user to adapt/extend the set of pre-defined model smells with further smells specific to his application scenario. The findings of the detector can subsequently be used for **report generation**.

- **Integration** of all aforementioned components into a research **framework** for the analysis of model-based software artifacts called the *artshop* framework.

- **Evaluation** of all aforementioned contributions with a set of **real-world artifacts** from academic and industrial case studies from the automotive, avionic and medical domain. This highlights the **scalability** and **applicability** of all techniques and analyses presented throughout this thesis.

1.4 Bibliographic Notes

Parts of this thesis were already published in publications authored or coauthored by the author of this thesis or were the results of bachelor's or master's theses that we supervised. In the following, we relate the chapters of this thesis to these publications/theses and detail the contributions of the author of this thesis.

The foundations described in Chapter 2 are mostly summarized from different sources that are cited accordingly. The summary of the characteristics of blocks and signals in MATLAB/Simulink models was initially created by the author as part of the work published in [59] and extended within the context of this thesis. The formalization of MATLAB/Simulink models introduced in Section 2.2.4 were partially contained in [59] and were created and later extended by the author of this thesis.

The incremental integration approach of model-based software artifacts described in Chapter 3 including the techniques describing the model repository, metamodel, tool adapters, graphical views, synchronization procedure and the subsequent use during static model analyses has been partially described in [57, 58]. The artshop metamodel and the import procedure of the tool adapter for MATLAB/Simulink models were created by the author of this thesis with minor contributions of various student research assistants. Julius Nehring Wirxel implemented the export procedure for the MATLAB/Simulink tool adapter as part of his bachelor's thesis [102]. An initial realization of the graphical viewer for MATLAB/Simulink models has been developed by René Rousseau as part of his bachelor's thesis [133] and was later extended by Stefan Schake and the author of this thesis. The tool adapter for IBM Rational DOORS was developed during the bachelor's thesis of Mirko Kugelmeier [87] including the model representation, the import/export procedures as well as the graphical views for these artifacts. We created the model representation for feature models and the generic import procedure described in Section 3.3.3 that was later extended for family and variant description models created with pure::systems pure::variants as part of the bachelor's thesis of Oliver Blasius [15]. The repository component and the synchronization mechanism were created by the author of this thesis.

The concepts for the feature derivation procedure and subsequent consistency check described in Section 4.3 and 4.4 were contributed by the author. A first implementation of these mechanisms was created as part of the bachelor's thesis of Oliver Blasius [15] and was later extended by Mirko Kugelmeier as part of his activities as a student research assistant.

Chapter 5 is based on ideas published in [59], which describes a first version of the signal reconstruction and flow-sensitive slicing approach for MATLAB/Simulink models. The concept, implementation and evaluation presented in this publication were created by the author. We further extend the techniques described in this publications by refinements of the overall signal dependency detection procedure and introduce further application scenarios using the reconstructed signal information as well as the implemented slicing algorithm. The proof-of-concept data dependency check for MATLAB/Stateflow Chart blocks described in Section 5.6, has first been implemented by Stefan Schake and was later refined by the author of this thesis. The slice-based cohesion and coupling metrics mentioned in Section 5.7.1 were implemented by Mirko Kugelmeier under the supervision of the author. The other extensions were created by the author of this thesis.

The techniques presented in Chapter 6 were initially developed during the bachelor's thesis of Stefan Schake [136]. The results of this thesis were published in [61] and later extended by the author. The concept for the repository guided cross clone detector proposed in Section 6.5 was created by the author, first implemented by Stefan Schake and later extended by the author of this thesis.

Parts of Chapter 7 were first published in [60], where an initial definition of the model smells for MATLAB/Simulink models was presented. They were developed in cooperation with Quang Minh-Tran from the Daimler Center for Automotive Information Technology Innovations (DCAITI) and Christian Dziobek from Daimler AG. The descriptions of the model smells from this publication were used for the formal definitions in Section 7.2. These definitions rely on the reconstructed signal information described as part of the flow-sensitive slicing approach presented in Chapter 5 and were created by the author. The refactoring operations described in [60], were created by Quang Minh-Tran as part of his research activities at the DCAITI. The EVL-based model smell detector and the report generator has been developed as part of the bachelor's thesis of Dennis Weir [159] and was later extended by Mirko Kugelmeier and the author of this thesis. The EVL model smell detection rules used during the evaluation of the model smell detector were created by the author.

In addition, all evaluations presented within this thesis were conducted by the author of this thesis.

Furthermore, we supervised the master's thesis of Norman Hansen who developed the initial version of a static value range analysis for MATLAB/Simulink models that relies on the model representations of the artshop framework [66]. The implemented techniques were later extended by joint work of Christian Dernehl and Norman Hansen [40, 41, 42, 43, 44].

1.5 Outline

The remainder of this thesis is structured as follows. Chapter 2 introduces the foundations and terminology of the processes, models, tools and concepts used throughout this thesis. Following that, Chapter 3 introduces the incremental integration approach for model-based software artifacts that the remainder of this thesis is based on, as well as the most important components of the *artshop* framework. The subsequent chapters describe the static intra- and inter-artifact analyses developed as part of the artshop framework. Chapter 4 outlines the feature derivation and subsequent consistency check developed for the variability documentation of a product line. After that, the signal reconstruction, dependency analysis and slicing algorithm are introduced in Chapter 5.

Following this chapter, Chapter 6 presents the clone detection and refactoring procedure for MATLAB/Simulink models. Chapter 7 describes the concept of model smells for MATLAB/Simulink models in relation to the common concept of code smells as known from traditional software engineering. Finally, Chapter 8 concludes this thesis by summarizing the results and discussing future work regarding the techniques presented throughout this thesis.

2 Foundations

This chapter introduces the foundations and terminology that are used throughout this thesis. First, we briefly introduce the concept of model-based software development in Section 2.1. After that, we describe the modeling tool MATLAB/Simulink and provide essential information regarding the syntax and semantics of its graphical block diagram notation as an example for an industrially used model-based development tool in Section 2.2. Section 2.3 introduces the concept of software product lines and the models used during its application in a software development process. Finally, further tools and their integration within industrial development processes are described in Section 2.4.

2.1 Model-Based Development of Automotive Embedded Software

In traditional software development, the main development artifact is represented as the source code of the system that is accompanied by various forms of documentation artifacts. Documentation artifacts include textual requirements or test specifications but may also include models that capture various views on the developed software system. Models can be used to describe the architecture of a software system, functional dependencies and interface or behavior specifications, but may become inconsistent to the source code they refer to, as updating is a time- and cost-intensive process [12].

In the last 20 years, the development processes for automotive embedded software have slowly shifted from traditional software development, with source code as its main development artifact, to model-based software development [12, 19, 20, 47].

2.1.1 Model-Based Software Development

In model-based software development, models represent the main development artifacts. These models are represented by a graphical notation, with defined syntax and semantics [12]. The key benefit of model-based software development in comparison to traditional software development is that domain specific concepts can be used to model the desired behavior of a program. These concepts are usually closer to the actual problem domain than the source code of programming languages used to program an embedded system [143].

Models are particularly useful if domain experts, such as engineers, are involved in the software development process, e.g. to provide input for control algorithms. Due to the abstractions that are offered by the modeling elements of a modeling language, even an engineer with no prior software development experience can create software by modeling a control algorithm using functional block diagrams, where typical operations from the engineering domain are available as basic model elements. Instead of using control and data structures like variables, loops or functions from traditional software engineering, an engineer can use a domain specific modeling language that he is proficient in and use constructs such as transfer functions, state charts and PID-controller [12].

Besides their descriptive nature, models can typically be executed in model-specific simulation environments, which enables early-stage testing and functional validation as well as subsequent reuse of validated model components [47]. Most modeling tools also include code generators that convert the *platform-independent* models [20, 143] into source code that is tailored to be executed in a specific hardware environment, representing a key criterion for the application of model-based development [137].

2.1.2 Model-Based Software Development in the Automotive Industry

One of the first industries to promote model-based software development intensively, was the automotive industry [30, 147]. While parts of a car have traditionally been developed independently from each other by a chain of suppliers and were more or less only assembled by the OEM, the innovation force of software forces OEMs to not only assemble the mechanical parts but also perform system integration of the software solutions developed by their suppliers. This highlights the demand for clear interface and behavior descriptions provided by the OEM to the supplier, as suppliers typically have a lot of freedom to realize individual solutions [113]. In addition, model-based development of automotive software is attractive due to its benefits in evolutionary development, i.e. the reuse and extensibility of existing functionality from previous system versions, and its platform-independent development [20, 143].

The benefits of model-based development in the automotive industry have been presented in a study discussed in [20]. Participants reported that a model-based development approach improved the communication regarding modeled system functionality, because of the use of domain specific modeling languages, e.g. function block diagrams. The study also highlights the benefits of the involvement of domain experts, which were not familiar with traditional software development but due to the use of the aforementioned domain-specific modeling language could contribute their know-how during system design. Other positive aspects include the application of rapid control prototyping enabled by early-state model behavior simulation, model verification techniques such as static analysis and model-in-the-loop tests.

To structure the interaction between supplier and OEM in terms of expected model formats, various company and domain specific norms and standards have been established, e.g. the ISO 26262 [73] or the AUTOSAR-Initiative [22]. These standards and norms significantly influenced currently applied development processes and the usage of consistent software architectures, which resulted in a nearly uniform domain-wide tool landscape [163].

Typical modeling tools used during model-based development of embedded software include AUTOSAR [22], MATLAB/Simulink [129], Modelica [124], LabView [125], ASCET [121] and SCADE [120]. Of these tools, MATLAB/Simulink is the defacto standard tool for functional behavior modeling in the automotive industry [47].

2.2 MATLAB/Simulink

The tool MATLAB from The MathWorks [127] integrates various algorithms to solve numerical problems and allows the graphical visualization of computation results. MATLAB features operations allowing matrix manipulation, function plotting, the creation of user interfaces and offers capabilities for user-specified algorithms.

Algorithms can be specified in the MATLAB programming language, also known as .m scripts, which offers access to a wide-array of built-in computation operations and can even be used to call libraries created in other programming languages such as *C, C++* and *Java*. Within the MATLAB programming language, all data available in MATLAB can be accessed and manipulated. Furthermore, there exist internal interfaces that allow external applications to interact with MATLAB by executing MATLAB code via remote method invocation.

While MATLAB is primarily intended to be used for numerical computations, its functionality can be extended by various toolboxes that are offered by The MathWorks and third party vendors. Toolboxes can either provide completely new functionality or further enhance existing functionality of MATLAB or even other toolboxes. Available toolboxes provide algorithms for image acquisition and processing, bioinformatics, statistics, symbolic computation or the graphical specification and simulation of control systems. The latter toolbox is officially called the Simulink programming environment and extends MATLAB by capabilities to graphically model control algorithms in the form of function block diagrams. In addition, Simulink also includes a parameterizable simulation environment, which offers multiple techniques to simulate the functionality of modeled diagrams with different sampling times.

The notation of a functional block diagram modeled in Simulink (also called *function-model*) is data flow oriented with most blocks representing atomic logical or mathematical operations connected by directed lines, which carry signals from one block to another. Other blocks structure a model by composing multiple blocks or lines into a single element and therefore reduce the visual complexity due to the introduction of hierarchic structures in a model.

An example for a MATLAB/Simulink model can be seen in Figure 2.1. The upper half of the figure shows the root of the model, while the lower half shows the content of a Subsystem block that introduces an additional hierarchy level within the model where further elements can be placed. A signal is routed from the Constant block in the root of the model into the Subsystem block, where it is first emitted by the Inport block In1 and subsequently multiplied with the value 42 by the Gain block. The resulting value is further propagated into an Integrator block that accumulates its received signal values

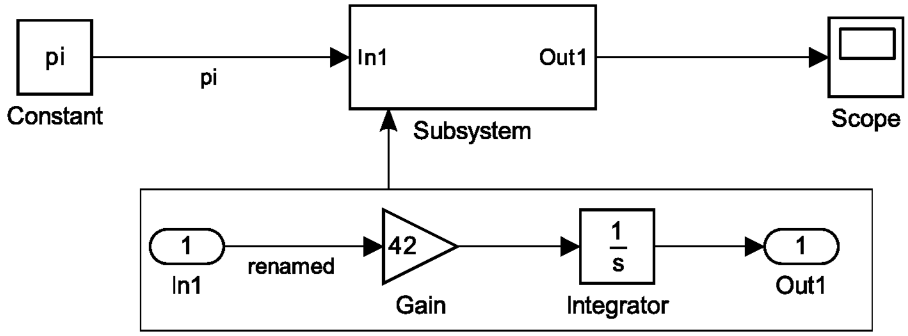


Figure 2.1: Example for a hierarchical simulink model

over time. The signal that is emitted by the Integrator block is finally routed back to the root of the model where it terminates at the Scope block.

While Simulink is shipped with a block set realizing logical and mathematical operations, additional blocks can be integrated by the use of various toolboxes. The MATLAB/Stateflow toolbox integrates statecharts into the blockset, which extend the Simulink modeling language by state- and event-based language constructs. This allows the modeling of state-based sequential decision logic via hierarchical finite automata.

Other toolboxes such as the *embedded coder* or *dSpace Target Link* extend Simulink with the capabilities for code generation. Functionmodels can then be transformed into C-Code that is then compiled for execution of a chosen target platform. The generated code for each block depends on its individual configuration.

In the next section, we will discuss the different block categories of MATLAB/Simulink and highlight the impact they have on the semantics of a Simulink model.

2.2.1 Blocks and Signals in MATLAB/Simulink

A block in a Simulink model is a processing unit that receives a set of input signals via its input ports (inports) that are transformed into a set of output signals emitted via its output ports (outports). The actual syntax and semantic of a Simulink block is defined by its type and block parameters. Blocks are provided to a modeler via so-called block libraries, which group blocks with similar functionality, e.g. logical, discrete or continuous operations. The type of a block defines its general appearance and the operation it performs. A block can further be configured by its block parameters by changing the semantics of its operation or its appearance. While certain parameters are shared among all Simulink blocks, such as the Simulink Identifier (SID), background color, position or name, some attributes are unique to certain block types like the *Gain* parameter of the Gain block shown in Figure 2.1.

Blocks in MATLAB/Simulink are connected by lines that transfer a signal from the outport of one block to the inport of one or multiple blocks. Signals carried by a line represent typed data values, e.g. (u)int8/16/32, double, float or fix-point numbers. These data values do not necessarily have to be scalar values but can also be a vector or matrix of values. The type of a signal is usually derived from the block that created it and can be influenced with the help of block parameters. By assigning a name to a line, its carried signal is also automatically named or assigned an alias if it is already named. Properties such as the name and data type inference rules for ports and lines are stored as parameters attached to the configured port or line.

While lines represent connections from one block to multiple blocks, depending on the block category of the destination block, signals might be propagated through multiple blocks and lines until they are consumed. Two of these block categories will be introduced in the following section.

Virtual Blocks vs. Nonvirtual Blocks

Simulink blocks belong to either of two categories, virtual or nonvirtual blocks. Nonvirtual blocks realize the semantic behavior of a model during its execution, such as the Gain block shown in Figure 2.1, which multiplies an incoming signal value with a term specified in the blocks parameters. Signals emitted from nonvirtual blocks are always new signals created as part of the transfer function of a block. In contrast, the main purpose of virtual blocks is the graphical organization of a model, i.e. signals received by a virtual block are not altered and the emitted signal of the virtual block is the same signal that entered the block. Examples for virtual blocks are the Subsystem and Inport blocks in Figure 2.1. A subset of the virtual block set is only considered virtual under specific conditions[1]. Depending on the configuration or the environment these blocks are placed in, they may become nonvirtual blocks and play an active role during the execution of a model.

Direct-Feedthrough vs Non-Direct-Feedthrough Blocks

Another way to categorize blocks is how their outputs are calculated. If the outputs of a block are directly calculated from one of its input signals, then this block is a direct-feedthrough block. However, if the output signal of a block is driven by a state of the block that depends on its input signals, the block is categorized as a non-direct-feedthrough block. An example for a direct-feedthrough block is a Gain block that multiplies its incoming signal by a value, while an Integrator block is a non-direct-feedthrough block as it internally accumulates all received signal values.

Bus-Capable Blocks

Bus signals can be used to reduce the visual complexity of a model by lowering the amount of lines visible on the screen or to trim the interface of Subsystem blocks in a model. There exist numerous ways to create bus signals in a MATLAB/Simulink model. The most common way to create and manipulate bus signals are the *BusCreator*, *BusSelector* and *BusAssignment* blocks.

[1] MATLAB/Simulink Web Help - Nonvirtual and Virtual Blocks: `http://de.mathworks.com/help/simulink/ug/nonvirtual-and-virtual-blocks.html`

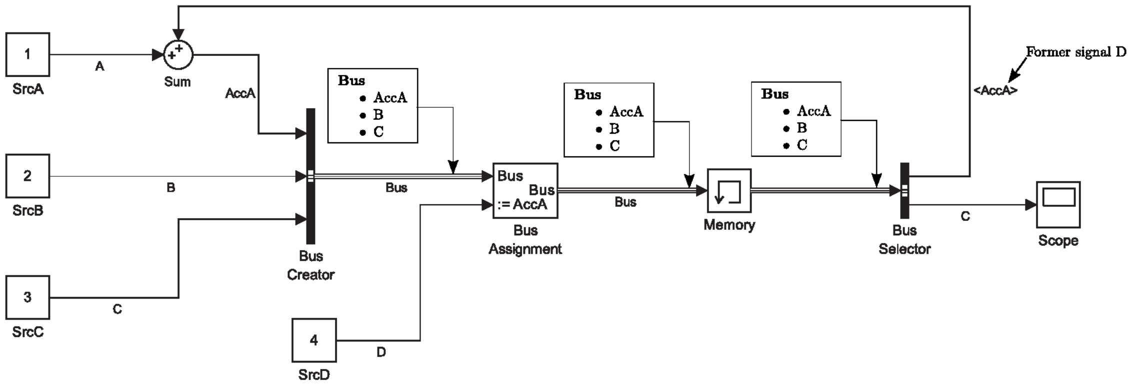


Figure 2.2: Bus signal creation/manipulation/resolution in MATLAB/Simulink (adapted from [59])

BusCreator The *BusCreator* block receives one or multiple input signals and hierarchically composes them into a bus signal emitted from this block as shown in Figure 2.2. Here, the signals *AccA*, *B* and *C* are composed into the bus signal *Bus*. A bus created by a BusCreator block must assert that no duplicate signal names are contained on the top level of the newly created bus, so that each signal can be identified explicitly by its fully qualified signal name ([bus name].[signal name]). The BusCreator block automatically renames all duplicate signal names by appending the string *'(signal x)'*, where x is the position of the renamed signal within the bus, to the name of a signal. Unnamed signals entering a BusCreator block are labeled in the same fashion.

BusSelector To extract a signal from a bus signal, a *BusSelector* block can be used as shown in Figure 2.2. This block can select and emit signals contained in a bus signal even if they are embedded in multiple hierarchically nested bus signals. Selected signals are identified based on their fully qualified signal name in the bus that has entered the block. Additionally, the block can be configured to select (output) the same signal multiple times and again create a bus signal from the selected signals to reshape the incoming bus signal. In the example, the signal name (*AccA*) is sufficient to select the signal with the corresponding name, as no additional bus hierarchy is present in the incoming bus signal. Signals that are not selected by the block are terminated, e.g. signal *B*.

BusAssignment The *BusAssignment* block allows the manipulation of a bus signal by replacing one or multiple signals in a bus signal with another signal with matching data type. The replaced signal is again identified by its fully qualified signal name in the bus signal entering the BusAssignment block. Signals that are replaced are terminated at the BusAssignment block, while the names of the replacing signals are aliased to match the replaced signals name. Further, the replacing signals position in the bus signal is also inherited from the replaced signal. In the example shown in Figure 2.2, the signal *AccA* is replaced by signal *D* and renamed to *AccA*.

While the BusCreator, BusSelector and BusAssignment block are virtual blocks, they also belong to the category of bus-capable blocks that are considered capable of handling bus signals[2]. These blocks can receive and/or emit bus signals. If a bus signal is propagated over a bus-capable block, it is continued with identical signal flow characteristics, i.e. bus structure, signal names etc. In addition to all virtual blocks, the set of bus-capable blocks also contains a set of nonvirtual blocks, which do not necessarily change the meaning of a signal. Examples for the functionality of nonvirtual bus-capable blocks are signal delay, signal characteristic checks or propagation of its incoming signals into a datastore or workspace variable. The *Memory* block shown in Figure 2.2 is an example for a nonvirtual, bus-capable block that temporarily delays its incoming signal. Signals that are propagated over a nonvirtual, bus-capable block are usually copied and treated as new signals by Simulink.

Modeling Hierarchy

While bus signals can be used to reduce the amount of lines in a model, the same concept can be applied to blocks by composing multiple blocks by the use of the *Subsystem* block. A Subsystem block can also have in- and outports and can be used as each other block in Simulink. Internally, a Subsystem creates a new hierarchical layer in a Simulink model where blocks, even Subsystem blocks, can be placed in. Signals are transported to this layer by so-called Inport and Outport blocks that propagate the signals received by the actual ports of the Subsystem block. This allows the creation of custom functionality and modularization of a Simulink model into smaller components with defined interfaces.

Masked subsystems One way to further model a Subsystem block as a distinct function, is to apply a so-called mask on it. A masked subsystem can be configured to change its visual appearance and provides a user interface where parameters can be specified that can be accessed by the blocks contained in the Subsystem block. In addition, parameters of distinct blocks can be *promoted* to be directly configurable via the configuration dialog associated with the masked subsystem. While unmasked subsystems can be navigated to via the Simulink IDE, masked Subsystem blocks need the user to enter a specific key combination to navigate to its content, which further establishes the masked subsystem as a self-contained function.

User-defined library blocks Subsystems can also serve as the basis for creating user-defined block libraries. Library blocks are stored in a so-called library repository managed by MATLAB/Simulink. This repository contains all default blocks available within the common blockset of MATLAB/Simulink. Once defined and populated with the desired functionality, a user-defined library block can be reused throughout one or multiple models by adding it via the library browser. Changes made to a library block are propagated

[2]MATLAB/Simulink Web Help - Bus-Capable Blocks: `http://de.mathworks.com/help/simulink/ug/bus-capable-blocks.html`

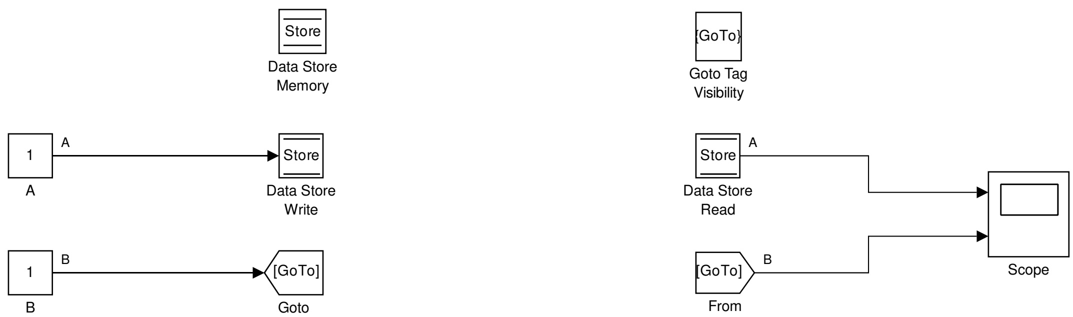


Figure 2.3: Examples for indirect signal flow using Goto/From and DataStoreWrite/Read blocks in MATLAB/Simulink

to all instances of the library block in a model. When combined with a mask, different instances of a library block can receive context-dependent parametrization.

Indirect Signal-Flow

As mentioned in the sections before, signals in Simulink are propagated over the lines attached to the ports of the blocks in a Simulink model. We already introduced one exception to this rule that describes the *indirect signal-flow* between different hierarchy layers of the Simulink models, which is not represented by a line in a model. There exist further modeling concepts that introduce indirect signal-flow within Simulink models. The *Goto* and *From* blocks that are displayed in the lower-half of Figure 2.3 can exchange signals between each other without being connected by a line. Goto and From blocks can even exchange signals over multiple hierarchy layers and from one Goto to numerous From blocks. These indirect connections are established via so-called goto tags, defined by a *GotoTag* block that defines a scope, which regulates where the goto tag associated with the block can be used. The scope can be local, i.e. only visible in the same subsystem, global, which means that it can be used in the complete model or scoped. The latter means that the tag is visible on the current hierarchy level and all hierarchy levels below it.

A similar concept that resembles the concept of global variables from traditional software engineering in Simulink are *DataStoreMemory* blocks. These blocks can store a passed signal value of arbitrary type for a given identifier and can be accessed and modified by the *DataStoreRead* and *DataStoreWrite* block displayed in the upper-half of Figure 2.3. Similar to the concept of scopes defined by GotoTag block, data stores, by default, can be accessed in a scoped fashion, meaning that it can be read/written from/to by DataStoreRead and DataStoreWrite blocks on the same hierarchy level or below. In contrast to Goto and From blocks, the actual values read from a DataStoreMemory may change throughout an execution cycle of a model as the read and write order depends on the execution order of the model. This might lead to read-before-write, write-after-read or write-after-write errors if the blocks are not correctly sequenced [9].

In the next section, we will introduce the concept of sorted order and control flow for Simulink models.

2.2.2 Model Simulation

One of the features of the Simulink toolbox is that the behavior of specified functionmodels can be simulated in an integrated simulation environment. Simulation is composed of three phases: model compilation, linking and the actual simulation loop[3]. The results of the simulation loop are computed by a so-called solver, which computes the states of the dynamic system expressed by a model at different time steps.

Model compilation phase Before a model can be simulated, it needs to be compiled into an executable form. During this phase, the model compiler, among others, evaluates the models block parameters to determine their actual values, calculates signal data types, performs block optimization, removes virtual blocks and calculates the *sorted order* of the model. The sorted order specifies the sequence of block invocation during a time step of the simulation. To express dependencies between the execution orders of blocks during simulation, MATLAB/Simulink uses so-called *execution contexts*.

Definition 2.1 (Execution Contexts in Simulink)**.**
An execution context contains a sorted list of blocks from a Simulink model that have to be sequentially executed during model simulation. It is associated with either the model (root execution context) or a block of a Simulink model and is entered once either the model or its associated block would be executed next. Once entered, all contained blocks and their associated execution contexts have to be executed before execution can continue in the parent execution context.

An execution context is called a conditional execution context if its execution depends on a condition specified in a block able to influence the control flow of a model.

Figure 2.4 shows an example model and its accompanying execution contexts. The sorted order of the blocks is annotated to each block and highlighted red. Most blocks of the shown model reside in the root execution context, while the Switch block creates two conditional execution contexts that are not executed until the condition of the Switch is evaluated. Depending on the result, either CEC1 or CEC2 is executed. After that, the signal computed by the respective execution context is propagated by the Switch block.

Typically, the sorted order cannot be influenced as it depends on the data dependencies between the blocks of an execution context. Nonvirtual blocks can be assigned priorities to indicate a relative order between blocks, which Simulink tries to honor during simulation [9]. Virtual blocks are not assigned a sorted order, as they do not influence the simulation.

As already shown in the example in Figure 2.4, a subset of the Simulink blockset actively influences which model parts are executed during simulation, by manipulating the control flow of a model. The following blocks can influence the sorted order and control flow of a model:

[3] MATLAB/Simulink Web Help - Simulation Phases in Dynamic Systems: `http://mathworks.com/help/simulink/ug/simulating-dynamic-systems.html`

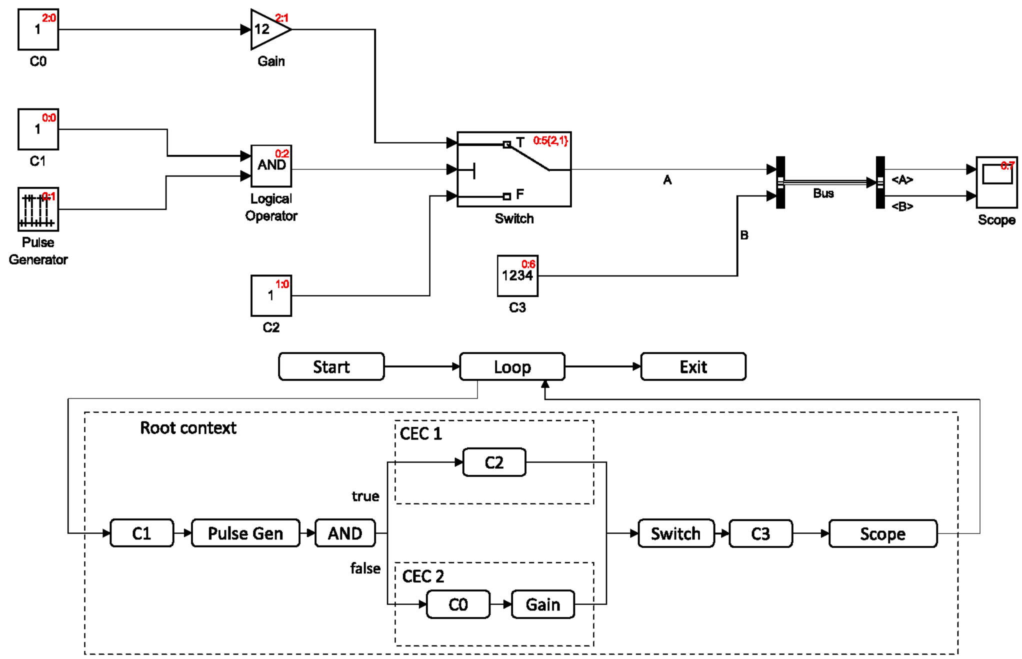


Figure 2.4: Execution contexts in MATLAB/Simulink demonstrated at an example model

- Conditional Subsystem blocks
- Loop Subsystem blocks
- Switch and MultiPortSwitch blocks

Conditional Subsystem blocks are defined to inherit one or multiple special inports whose input determines if the blocks contained in the conditional subsystem are executed. This can be dependent on a rising or falling edge received on these special inports or driven by an action signal that is emitted from a Switch-Case or If-then-else block. Conditional subsystems are said to be *atomic*, meaning that they are assigned a new execution context, which again contains all nonvirtual blocks of the conditional subsystem.

Loop subsystems are also considered to be atomic and trigger the execution of the subsystem as long as a condition defined in the block evaluates to true or for a set amount of times [116].

Switch and MultiPortSwitch blocks allow the propagation of one of their input signals based on a condition that is specified with respect to the signal value of a control port of the block. If the simulation environment is configured to allow conditional execution behavior, the evaluation of the control port of a Switch block precedes the blocks actually computing the signal propagated by the block in the sorted order. Based on the evaluation result of the conditional property controlling the selection of the propagated signal, the simulation environment can entirely skip the computation of the signal that is not propagated by the Switch block as shown in Figure 2.4.

Linking phase During the linking phase, MATLAB/Simulink allocates memory for the signals, states and run-time parameters of the model and initializes them according to the parameters given in the model.

Simulation loop During the simulation loop, the simulation environment successively executes all blocks in the sorted order computed in the model compilation phase to compute the model's state and outputs for a given time horizon and resolution. Consecutive executions of the model are called time steps and changes to states and input/output values in time step t_n are propagated to time step t_{n+1}. In each time step, the simulation environment performs the following actions:

- **Compute the model's output** In this phase, all block outputs are calculated by executing all blocks in the sorted order of the execution contexts.
- **Compute the model's state** Based on the outputs calculated in the previous steps, stateful blocks compute their new state.
- **Check for discontinuities in the continuous states of blocks** An optional step only executed if a variable time step solver is used. During this step, a technique called *zero-crossing detection* is used to detect discontinuities in the variables (outputs/states) of the model and adaptively change the step-size to increase the accuracy around the discontinuity, as variables change rapidly in the vicinity of a discontinuity.
- **Compute the time for the next time step** In the last step, the time of the next time step is calculated.

2.2.3 MATLAB/Stateflow

A toolbox that expands the functionality and the modeling notation of Simulink is MATLAB/Stateflow [130]. MATLAB/Stateflow is an environment that allows the modeling of discrete controllers using hierarchical state-machines in combination with flowchart diagrams. The Stateflow modeling environment is completely integrated within MATLAB/Simulink and can be used by placing a so-called Chart block in a Simulink model. By navigating into the Chart block, statechart semantics can be used to model the internal behavior of the Chart block as it can be seen in Figure 2.5 that contains the statechart contained in the Simulink example model sldemo_fuelsys.

A stateflow diagram, also called statechart, consists of

- **States** An active state captures the current state of the statechart. Can be enhanced with operations from MATLAB or C that are executed on certain state-action, e.g. when the state is entered, active or exited. States can again contain other states.
- **Transitions** Connects two states and/or junctions with each other representing the transition of the system from one state to another. Can be enhanced with a

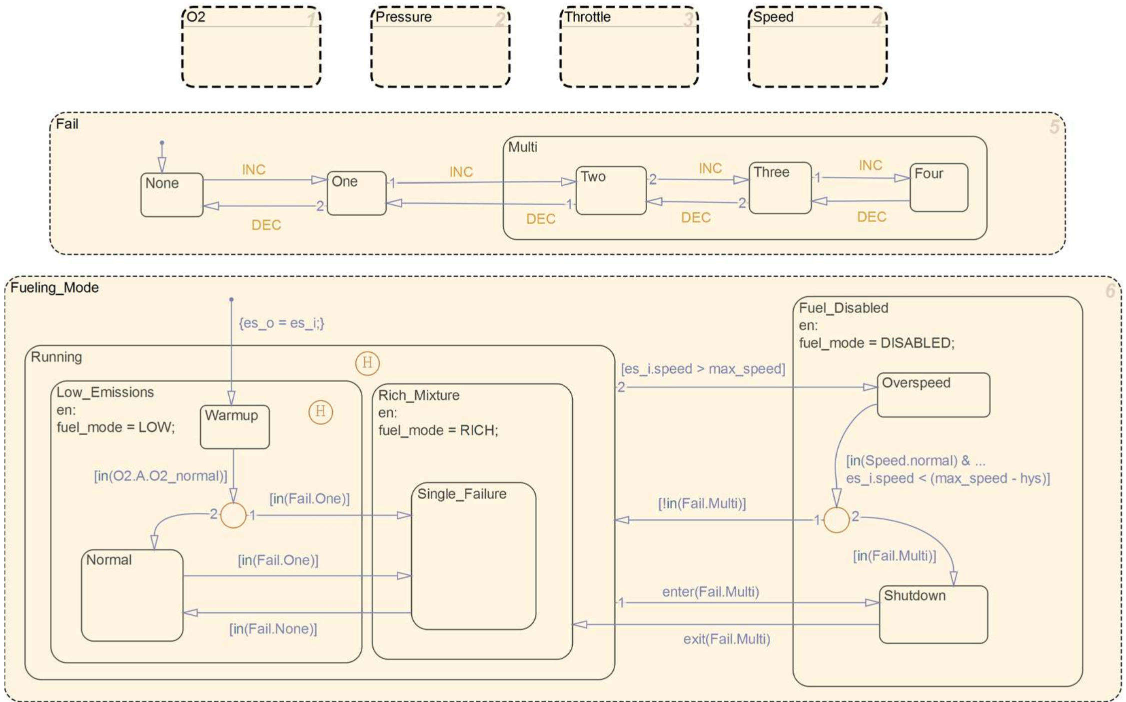


Figure 2.5: Stateflow chart contained in the example model sldemo_fuelsys

label that may specify a guard that has to be satisfied before the transition can be taken, a condition action performed once the guard is satisfied and/or a transition action that is executed once the final transition destination has been found to be valid. Syntax of these actions again are either MATLAB or C.

- **Junctions** A junction may be used to represent multiple different transition paths for a single transition. They may be placed as an endpoint of a transition and can spawns one or more new transitions with different destinations. Each of these transitions again may have guards and actions attached to them.
- **Functions** Can be used to add user-specified functions in the form of flow chart algorithms, MATLAB functions, Simulink functions, truth tables or custom code. Defined functions can be called from state- or transition actions.
- **Variables** Intermediate results of state and transition actions can be stored in variables associated to the chart. They are also used to access the signals received by a Chart block in Simulink, as a corresponding variable is created for each input and output signal received/emitted via the ports of the Chart block.

Further information about the syntax and semantics of Stateflow can be found in [132].

2.2.4 Formalization

To ease the creation of definitions regarding certain structures of Simulink models throughout this thesis, we introduce a formalization of Simulink models in this section.

Definition 2.2 (MATLAB/Simulink Model)**.**
A MATLAB/Simulink model $\mathcal{M}$ is defined as

$$\mathcal{M} = (B, P = P_{in} \cup P_{out}, L, I, f_p : P \to B, h : L \to I)$$

The set B denotes the set of blocks contained in the model $\mathcal{M}$. Each block is associated with a set of in- and outports from $P = P_{in} \cup P_{out}$, that are mapped to their corresponding block, by the mapping function f_p. Blocks can be connected by a line $l \in L$ that carries a signal from one specific outport of a block to one or multiple inports of one or multiple blocks. Lines are labeled by the labeling function h that assigns each line an atomic signal name from the set of signal names I. For the sake of simplicity we assume that each line in L connects exactly one port from P_{out} with one port from P_{in}, which is represented by the graph $G = (P, L)$. Thus, branches in lines are resolved by creating a distinct line for each destination port of the initial line.

To express signal relationships created by implicit signal flow (see Section 2.2.1, we use the notation of virtual lines/ports first introduced by Merschen et al. in [97]. Virtual lines and ports are added to blocks connected by indirect signal flow and are included in the set of lines L and ports P.

A block of a MATLAB/Simulink model $\mathcal{M}$ is further defined as follows.

Definition 2.3 (Blocks in MATLAB/Simulink)**.**
A block b from a MATLAB/Simulink model $\mathcal{M}$ is defined as $b = (l, t, P_b, A, b_P)$ with

- *Block label l*
- *Block type t*
- *Set of ports P_b*
- *Set of attributes A*
- *Block parent b_P, $\perp$ if contained in the root of the model.*

To allow the differentiation between different types of blocks, the block type t stores the block type as a string. Moreover, each block contains a set of attributes A that correspond to its block parameters. Each attribute is represented as a key-value pair. For the sake of simplicity, we refrain from further defining individual attributes but directly refer to them as needed.

During simulation, the sorted order computed by the simulation environment determines the order of execution of nonvirtual blocks in a time step of a model.

Table 2.1: Functions for the navigation and exploration of a formalized MATLAB/Simulink model $\mathcal{M} = (B, P = P_{in} \cup P_{out}, L, I, f_p, h)$

Relation	Logical signature	Definition
Block parent	$B \to B$	$parent(b = (l, t, P_b, A, b_P)) = \begin{cases} b_P, & \text{if } b_P \neq \perp \\ \perp, & otherwise \end{cases}$
Direct block containment	$B \to \mathcal{P}(B)$	$cssys_{dir}(b) = \{b_c \mid b_c \in B \land parent(b_c) = b\}$
Recursive block containment	$B \to \mathcal{P}(B)$	$cssys_{rec}(b) = \bigcup_{b_c \in cssys_{dir}(b)} \{b_c\} \cup cssys_{rec}(b_c)$
Block ports Block inports Block outports	$B \to \mathcal{P}(P)$ $B \to \mathcal{P}(P)$ $B \to \mathcal{P}(P)$	$ports(b = (l, t, P_b, A, b_P)) = P_b$ $ports_{in}(b) = ports(b) \cap P_{in}$ $ports_{out}(b) = ports(b) \cap P_{out}$
Block successor	$B \times B \to \mathbb{B}$	$succ(b_1, b_2) = \begin{cases} true, & \text{if } \exists\ (p_1, p_2) \in L : \\ & \quad p_1 \in ports_{out}(b_1)\ \land \\ & \quad p_2 \in ports_{in}(b_2) \\ false, & otherwise \end{cases}$

Definition 2.4 (Sorted Order of Nonvirtual Blocks in MATLAB/Simulink)**.**
Let $B_{Nonvirtual}$ be the set of nonvirtual blocks of a MATLAB/Simulink model $\mathcal{M}$. The sorted order between the blocks of $B_{Nonvirtual}$ is captured by a linear order $\leq_{SO}$. If $a \leq_{SO} b$ holds with $a, b \in B_{Nonvirtual}$, then block a is executed before block b.

Each block contains a set of ports containing in- and outports that are used by the block to receive and emit signals to its neighboring blocks.

Definition 2.5 (Ports in MATLAB/Simulink)**.**
A port p from a MATLAB/Simulink model $\mathcal{M}$ is defined as $p = (n, l, b_p)$ with

- *Port number n*
- *Port label l*
- *Parent block b_p*

The definitions provided in this section and the functions shown in Table 2.1 will be used for the formal definition of properties and characteristics of MATLAB/Simulink models throughout this thesis.

2.3 Software Product Lines

A characteristic problem OEMs from the automotive industry are facing is that they need to develop multiple automobile models as fast as possible, while at the same time ensuring high product quality. In addition, each car can be configured individually by the customer and has to satisfy legal restrictions imposed by the market it is sold in, which results in a multitude of functional variants of an automobile. As most customer configurable functions are software-based, the variability within an automobile also implies variability in software [46]. An established method to pool and manage multiple variants is the use of so-called Software Product Lines (SPL) [28]. SPLs promote systematic reuse of software artifacts, by creating the common and variable properties of all variants of the software artifacts in an SPL. Individual variants, also called products, can then be derived from this SPL.

Definition 2.6 (Software Artifact)**.**
A software artifact A is an element created during a software development process and may refer to architecture specification, requirements, test cases, source code or models.

Definition 2.7 (Variant [110])**.**
A variant, also called product, of a variable artifact represents an instance of this artifact, which is distinguishable from all other instances of this artifact by at least one property.

A property may refer to software functionality, e.g. a driver assistance or navigation system, but can also refer to physical properties of the product, e.g. sensors, engine type or infotainment systems.

While SPLs provide benefits during the development and maintenance of software variants, additional effort has to be invested to create, maintain and manage the complexity that is introduced by the SPL and its contained artifacts during *Software Product Line Development*.

2.3.1 Software Product Line Development

Software product line development introduces two essential principles that distinguish this process from single product development [16]:

- Analysis and modeling/documentation of the variability of the product line (variant management)
- Separating development activities into domain and application engineering

The aspect of variant management addresses the necessity of continuous analysis and documentation of the communal and variable properties within all artifacts of the product line. This ensures the availability and consistency of the variability documentation across all artifacts of the product line during all phases of the development process. The variant management method used throughout this thesis will be introduced in Section 2.3.2.

The second aspect addresses how the artifacts of the product line are developed and used to derive individual variants. These development activities are logically separated

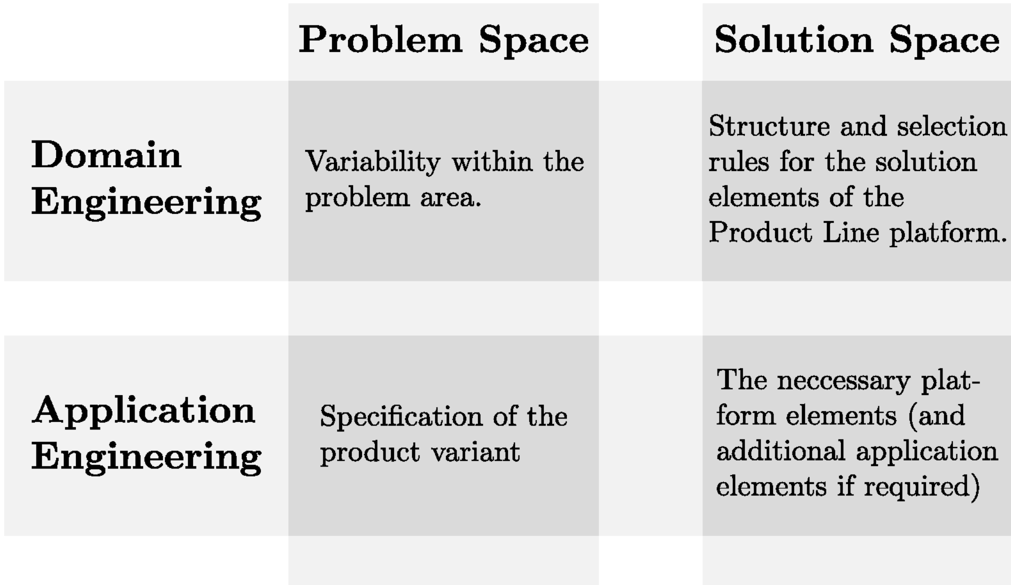


Figure 2.6: Activities of SPLE (based on [37])

into domain and application engineering [110, p. 20-21]. A core concept of software product line development is the use of a so-called platform, which represents a set of reusable software artifacts. The development of the platform and the derived products are typically logically separated.

- **Domain engineering** During this process, the platform artifacts of the product line are developed including all commonalities and variability of the product line.
- **Application engineering** In this process, a product is derived from the defined platform artifacts by instantiating a certain variant and ensuring its correctness according to the needs of the derived product.

Variant instantiation is performed by selecting a subset of the platform artifacts and configuring them to fit the need of a specific software product variant [37].

The activities of domain and application engineering can further be divided by separating both processes by activities concerning the problem and the solution space as it can be seen in Figure 2.6 [34, 37].

Problem Space In the problem space, the development process starts with a specification of the desired common and variable properties of the platform artifacts. This specification also includes documentation of inter-dependencies or constraints between the variable properties of the product line on the domain engineering level. A product variant on the application engineering level is then described by a product configuration including a combination of variable properties that were defined in the variability documentation of the problem space.

Solution Space The solution space describes the instantiation of the common and variable properties of the problem space as actual development artifacts of the platform. These artifacts are developed during domain engineering where they have to be linked to the properties from the problem space by the use of the variability documentation. Application engineering finally includes the derivation of a product variant, based on a given product configuration from the problem space, which is used in combination with the links of the variability documentation.

One approach to model variability in the problem and solution space are variation points that can be defined in the variability documentation of the artifacts of a product line.

Definition 2.8 (Variation Point[110]).
A variation point is a location within the variability documentation, which documents the possibility to select one or multiple variable properties of the product line for different product configurations. These variation points are linked and instantiated in the artifacts of the platform of the product line, to allow configuration-based product derivation during application engineering.

Modeling variation points and tracing them to their respective instantiations in the solution space is a fundamental part of variant management [10]. Many different methods of variability modeling and management have been established in the literature: feature models proposed by Kang et al. [76], orthogonal variability model by Pohl et al. [110] or delta modeling proposed by Schaefer et al. [135]. In this thesis, we will use the *feature modeling* approach by Kang et al. [76] to document variability in the problem space that will be introduced in the next section.

2.3.2 Modeling Variability

One approach to model variability in the problem space is the *Feature-Oriented Domain Analysis* (FODA) approach introduced by Kang et al. [76, p. 36], which proposes the use of so-called feature models. Features contained in a feature model represent the properties that products of a software product line may contain.

Kang et al. divide the set of features contained in a feature model in three categories:

- **Mandatory features** These features represent the commonality of the artifacts of the product line. They are present in every product derived from the product line.
- **Optional features** Optional features may be, but do not necessarily need to be part of a product.
- **Alternative features** These features typically appear in groups of features of which exactly one has to be selected as part of a product.

Further structures exists that enable the modeling of feature interdependencies. Some features might *require* the presence of other features or may only be present if certain other features are *excluded*. These require and exclude relationships may also be modeled

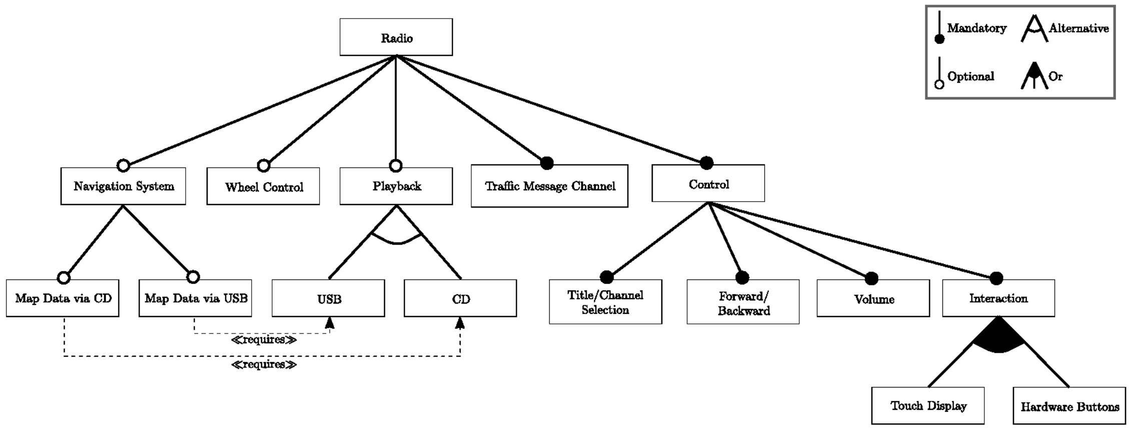


Figure 2.7: Example feature model (adapted from [161])

in a feature model. For example, a driver assistance system might require the presence of certain sensor packages within a car, while a certain series might exclude the configuration of a low class sensor package, because its base-line features already require the high-class sensor package. Thus, a downgrade is not possible.

Feature models have a tree-like structure with the root feature representing the product itself that is decomposed into the subsystem it contains. Figure 2.7 shows an example of a simple feature model for a radio that was adopted from [161]. This example includes mandatory as well as optional features. Furthermore, the features that are subordinated to the *Playback* feature are part of an alternative feature group of which exactly one has to be selected.

Since its initial introduction, several extensions have been proposed to the initial notation by Kang et al. Notable example are the introduction of *Or features*, which represent a group of features of which at least one has to be selected [34, p. 91-94] (see *Interaction* feature in Figure 2.7) or the introduction of cardinalities, which describe how many child features can be selected from a given parent feature [33].

2.4 Industrial Application and Tools

In addition to the management overhead introduced by software product line development, software for electronic control units (ECU) is developed using company-specific development processes.

In the following, we will introduce an exemplary software development process from the automotive industry and describe additional tools that are used during this development process.

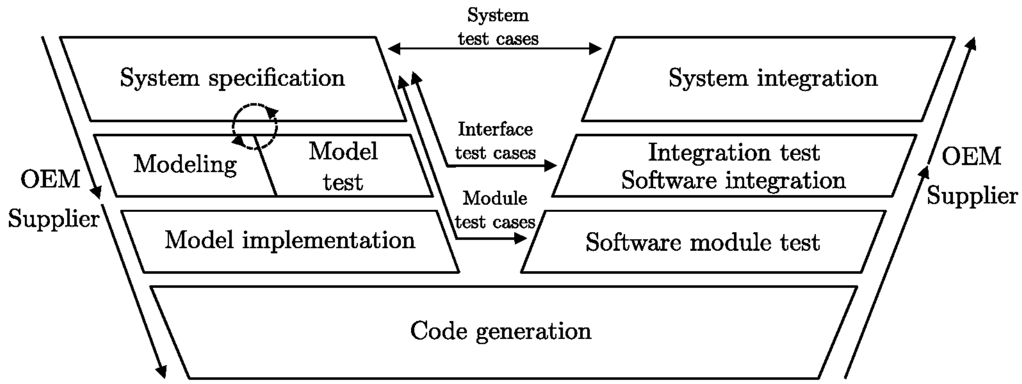


Figure 2.8: Model-based design process at Daimler AG (adapted from [100])

2.4.1 Development Process

The development process of model-based automotive software is typically structured as a V-model. Instead of performing the process steps of the development process in a linear step-by-step fashion, the V-model introduces inter-relationships between process steps, which promote the subsequent validation of early-stage development artifact. Figure 2.8 shows an exemplary model-based software development process used at the Daimler AG [100]. The process starts with the system specification, which includes variant management in the form of feature modeling and the specification of requirements that correspond to the modeled features that the system is composed of. The resulting requirements specification is used to create a MATLAB/Simulink model of the system including all common and variable features of the variability documentation. The requirements specification also serves as a reference document for the definition of the test plans for all test phases of the development process, which can be created before the artifacts to be tested are developed. Thus, a test plan in the form of model-in-the-loop tests is developed in parallel to the design of the functionmodel in MATLAB/Simulink and later used to systematically test and refine the created behavior model.

The artifacts created up to this point are then handed over to a supplier for implementation and ECU integration. The supplier implements the model, generates the code for the implemented models and performs software module tests in the form of software-in-the-loop tests on the target ECU. These tests are again derived from previously created model-in-the-loop tests and subsequently refined/extended.

After these steps the ECU and the integrated software is again handed back to the OEM, where component (component hardware-in-the-loop) and system (system hardware-in-the-loop) tests are defined and carried out. Finally, the process concludes with the integration into the physical vehicle and respective integration tests.

Various tools are used to create and manage the software artifacts during the different stages of the development process. Besides MATLAB/Simulink, notable tools in the context of this thesis are IBM Rational DOORS and pure::systems pure::variants. IBM Rational DOORS can be used to create and manage requirement or test case specifications, while also managing the traceability links between the elements of these specifications.

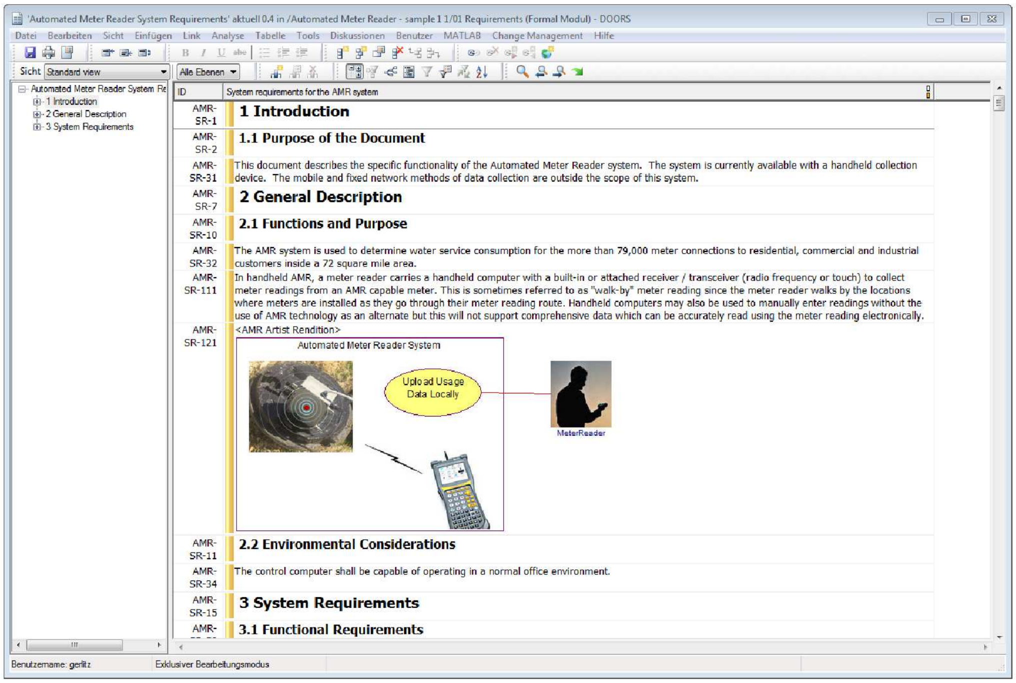


Figure 2.9: Example of a formal module in IBM Rational DOORS

The variant management tool pure::variants is used to create and manage the variability documentation in both problem and solution space. Both tools will be introduced in the next section. Further information about the described development process at Daimler AG and the tools that are used during this process can be found in [46, 100].

2.4.2 Tools

Besides MATLAB/Simulink, two other tools were mentioned in the previous section that are commonly used during automotive software development processes: IBM Rational DOORS and pure::systems pure::variants. These tools will be described in the following sections.

IBM Rational DOORS

IBM Rational DOORS (Dynamic Object Oriented Requirements System) is a requirements management tool developed by IBM. It is a client-server application and saves application data in a proprietary database. Users login to the server via the client application and can collaborate on the data available in the database. Application data mainly consists of so-called *formal modules*, which contain objects associated with user-defined and -selected attributes. Each object of a formal module is assigned a unique id – to enable requirements traceability and to track the object during its lifecycle – and can contain child objects. Within the client application, a formal module is typically represented as a table with rows representing the objects of the module and columns representing the attributes of the object as it can be seen in Figure 2.9. The object hierarchy of the module is flattened in the tabular view but still available in the navigation pane on the left-hand side. Attributes of objects, i.e. cells of the table, can contain data of varying data types. Besides text, integer, floating-point numbers and images, DOORS allows the storage of OLE (Object Linking and Embedding) objects, e.g. excel diagrams or word documents, RichText and user-defined enumerations.

IBM Rational DOORS further supports the creation of traceability links between objects of the same and/or different formal modules. Links can be analyzed, visualized in a graphical tree view and used for navigation between linked objects.

Formal modules can be organized within projects and folders and restricted for single or groups of users via a role management system that is available for the administrators of the server.

The functionality of the client-application can be extended by the use of the DOORS Extension Language (DXL). The DXL allows the creation of additional elements in the graphical user interface and the specification of user-specified analyses on the data stored in the database. DXL is an imperative programming language resembling C or C++ and includes typical control structures and primitive datatypes, e.g. bool, char, int, real, but also complex datatypes such as strings, structs and types for tool internal structures as the aforementioned modules, objects, folders and projects. More information on the syntax and available API in DXL can be found in the DXL reference manual [122].

As certain base functionality, e.g. import/export functions and impact analyses, are already implemented in DXL, the integration of DXL in DOORS resembles the integration of the MATLAB programming language in MATLAB.

pure::systems pure::variants

The tool pure::variants from pure::system is a variant management tool that supports developers during the lifecycle of a software product line. It supports the creation of variability models in the problem space using the notation of feature models as introduced in Section 2.3.2. In addition, the tool allows to trace features from their definition in the problem space to their realization and corresponding variation points in the solution space, by the use of *component family models*.

Family models were first proposed by Beuche et al. in [13] to capture the formal relationship between the variability documentation in the problem space (feature model) and the solution space. A family model consists of components that describe the internal structure of an artifact of a product line and their relationships to the features of the problem space. Components contain functionality that realize one or more features from the product line and may be decomposed into further components and elements specifically targeting elements and logical structures from the solution space [13]. Feature restrictions can be specified for all elements of the component model, which represent the traceability links connecting the elements from the problem with the solution space. It is also possible to generate a family model based on feature annotations in the artifacts from the solution space and synchronize these annotations with a created family model by the use of available tool adapters. For example a family model for a formal module in IBM Rational DOORS might contain feature annotations that can be synchronized to the corresponding family model in pure::variants.

pure::variants further allows the specification of specific product variants on the basis of feature models by the use of so-called *variant description models*. A variant description model describes the set of features that form a specific product in a product line. Violated feature restrictions imposed by requires and excludes relationships among features are highlighted by the tool to prevent the derivation of incorrect variants. The tools further supports the configuration of variation points in the artifacts of the solution space, based on a given variant description model. For this operation, various tool/artifact adapters are available that realize the actual application engineering within the artifacts of the solution space. As a result of the derivation process, copies of the platform artifacts are generated that only contain the elements that are associated with the features of the variant description model.

3 Incremental Integration of Model-Based Software Artifacts

During the phases of a model-based software development process, different tools are used to create the software artifacts that capture the various characteristics of software, e.g. requirements, architecture, test specifications or behavior models. These tools are usually isolated from each other, only focus on particular phases of the development process and target a specific engineering domain [19]. While most tools offer interfaces that can be used to access or manipulate tool-specific artifact data, these interfaces are not standardized between tool vendors and inter-tool information exchange via these interfaces is realized on a case-by-case basis. In addition, each tool uses unrelated and often proprietary model representations that are isolated from each other, which further complicates traceability and documentation across tool borders. Consequently, traceability information and documentations are scattered across all artifacts of the development process. This is further aggravated if the evolution of model data also has to be taken into account, as not all tools support the storage of revision information in their artifacts.

Another problem imposed by proprietary model representations is that model analyses used during verification and validation phases are limited to the analyses supported by the tools themselves. While some tools can be extended by the use of included scripting languages or even by a plugin architecture, analyses need to be customized for each tool environment. Depending on this environment, certain analyses might not be applicable at all, as the resources or performance of the programming language provided for user-defined analyses are not sufficient. Additionally, time and effort has to be invested to learn and apply the concepts needed to extend a proprietary tool/framework. Visualization of analysis results is also limited to the API available within the respective tool environment.

To be able to perform tool-independent consistency and static model quality analyses across tool borders, a model repository is needed which stores artifact information in a uniform way and preserves change information across artifact revisions. This enables the uniform creation of model analyses targeting artifacts from different tools, which can then be implemented against the interface of the model repository.

3.1 Overview and Outline

In this chapter, we first discuss concepts for an *incremental integration* process for model-based software artifacts and then show how they have been realized in the artshop framework. During artifact integration, artifacts from different sources are transformed into a tool-independent representation based on a common metamodel and are integrated into a joined representation, usually stored in a model repository [95]. Traceability information and documentation can directly be annotated to the integrated artifacts and their elements but should be logically separated from actual model data to prevent the entanglement of model and meta-data in the joined representation. As artifacts are constantly evolving outside the model repository, synchronization mechanisms are needed to incrementally integrate changes of the source artifacts into the artifacts of the model repository. By applying changes incrementally to the current revision of the integrated representations, modeled traceability links and documentation between/on integrated artifacts and their elements are preserved. This also enables the tracking of changes to an artifact on a structural level, which is typically not possible in file-based version control systems that only track changes in the binary representation of a file.

Thus, to realize incremental integration of model-based software artifacts the following concepts need to be addressed:

- **Metamodel** This metamodel is shared across all integrated artifacts and provides common concepts present in most source artifacts. It also shall provide strictly separated concepts for model data and meta-information to prevent the entanglement of data from different information sources.
- **Tool adapter** Provide an interface to a specific tool by providing a model, derived from the common metamodel, that represents an artifact from the tool and an import process that can instantiate this model based on a source artifact from that tool.
- **Model repository** The repository is used to manage artifacts imported by the tool adapters, i.e. storing and providing access to artifacts, as well as managing the history of stored artifacts.
- **Synchronization** A method to synchronize the artifacts stored in the repository with its source artifacts.

These components have been realized as part of the *artshop framework*, a client-server application that consists of a client *Eclipse E4 RCP* application while the server is based on the Connected Data Objects (CDO) [119] framework as the repository backend. It has been developed to support the described incremental artifact integration process and as a research platform for the development of model analysis techniques for the supported artifacts.

3.1.1 The artshop Framework

Figure 3.1 shows an abstract view of the architecture of artshop. The framework is partitioned into 4 components responsible for distinct tasks:

- **artshop.core** The core component supplies the metamodel that is needed for the derivation of specific artifact representation models. Furthermore, it manages results

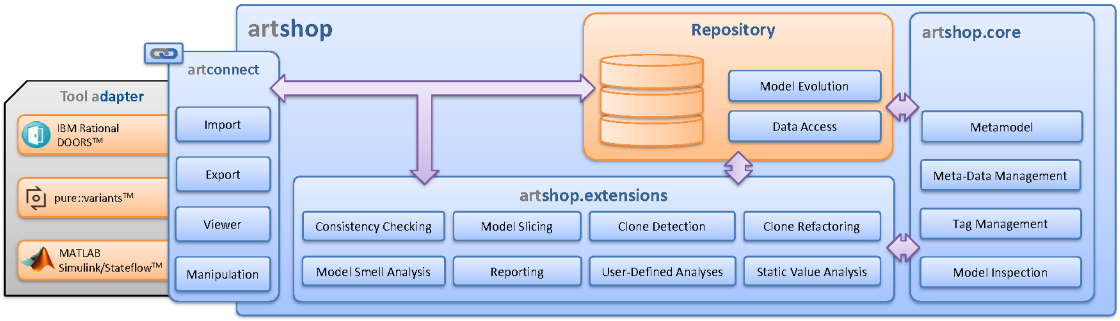


Figure 3.1: Architecture of the artshop framework (adapted from [57])

detected by the analyses of the framework as well user-created meta-information, e.g. comments or traceability links.

- **artconnect** The *artconnect* component encapsulates tool adapters for three modeling tools and provide interfaces for the import and export of model data. These tool adapters also provide the data structures based on a specialization of the metamodel supplied by the artshop.core component and may register artifact specific views that are used for visualizations. Tool adapters are added via plugins and automatically register their data structures and operations in the application.

- **Repository** This component is responsible for the storage and synchronization of model data extracted via the artconnect component, as well as all further data introduced by analyses of the framework. In addition, it manages the history of elements during their evolution and can execute queries against data stored in the repository.

- **artshop.extensions** The *artshop.extensions* component is freely extensible and can use all operations and services offered by the other components to realize analyses, visualize their results, derive and store traceability links or provide services such as report generation to other components of the framework. Again, extensions can be provided via plugins that automatically register provided operations in the graphical user interface of the framework.

The rest of this chapter is structured as follows. First related work regarding artifact integration is discussed in Section 3.1.2. After that, the realization of the aforementioned concepts as part of the artshop framework are introduced, including the technologies used during this process. The metamodel of the artshop framework is presented in Section 3.2 and utilized throughout Section 3.3, where concrete instantiations of the metamodel are discussed in the context of the artshop tool adapters. The technologies and techniques used to realize the model repository and its accompanying synchronization mechanism are described in Section 3.4. In Section 3.5, an evaluation of the import/export procedures of the tool adapters and the performance of the synchronization mechanism is presented. Finally, the chapter is concluded in Section 3.6.

3.1.2 Related Work

The concept for artifact integration used in this chapter is based on the concept presented in the PhD thesis of Merschen [95], which proposes the use of tool specific adapters to import model data and store them in a database. The PhD thesis of Merschen focuses on non-incremental artifact integration and therefore does not address artifact synchronization mechanisms and artifact revisions in the model repository. The tool adapters described by Merschen can only import a fraction of the data actually available within targeted tools, which limits the scope of analyses. As we will show during the evaluation in Section 3.5, the tool adapters presented by Merschen miss up to 99.96% of available tool data in the case of MATLAB/Simulink models. Merschen proposes the use of so-called virtual lines/ports, which represent connections between elements of a MATLAB/Simulink model connected by indirect signal flow (see Section 2.2.1). Virtual lines are calculated during model import and added to the model to ease the application of model analysis techniques. An extended version of this technique has been developed in the context of this thesis. Additionally, the annotation concept presented by Merschen, which was also discussed in [96], proposes the entanglement of model data with meta-information, which is avoided by a strict separation of model data and meta information in the artshop metamodel.

Broy et al. [19] describe an ideal approach to solve the problems introduced by the isolated tool landscape in the domain of model-based software engineering by introducing a pervasive modeling theorem across all phases of the development process.

Artifact integration was also targeted by the MESA project [51] as part of an effort to support tool independent conformity analyses on MATLAB/Simulink models and formal modules from IBM Rational DOORS. The approach of MESA is similar to the one presented in this thesis, as a metamodel is defined for the targeted modeling languages based on the MOF Standard [126]. In case of MATLAB/Simulink, the model is instantiated by a tool adapter that directly communicates with MATLAB via a COM-Interface to extract model data. Extracted models are stored in a model repository generated by a proprietary tool based on the created metamodel. Analyses can then be performed on the models stored in the repository, which are conformity and consistency checks based on OCL constraints. Unlike the solution presented in this thesis, meta information are not considered as part of the model and inter-artifact relationships are derived based on shared identifiers.

The MATE project [91, 148] also aims to implement tool independent analyses and model transformations for MATLAB/Simulink models. As in the MESA project, a tool-independent meta-model based on MOF 2.0 [126] is defined using MOFLON, that can either be instantiated by directly communicating with MATLAB/Simulink via a tool adapter or by a model parser that instantiates a model based on the binary model files created by MATLAB/Simulink. MOFLON is a meta-CASE-Tool [4] that allows the generation of a model repository to store created model instances. All analyses are performed on the instances of the metamodel, which can either be obtained from the repository or generated in an online-fashion via the mentioned tool adapter.

Besides the MESA and MATE projects, several other approaches have been presented to ease the creation of analyses of MATLAB/Simulink models as the API provided by Simulink to access its model data is complicated and not suited to implement complex analyses. The *Simulink Library for Java* developed by CQSE GmbH supports the import of Simulink models directly from supplied model files. It includes a metamodel for the representation of MATLAB/Simulink and Stateflow models and can instantiate these models by directly parsing supplied MATLAB/Simulink model files (.mdl/.slx). As the importer is not backed by information from the Simulink runtime, no compile-time attributes, e.g. signal dimensions or data types, are available in imported models. In addition, model libraries cannot be dynamically fetched as it is the case for models imported via the MATLAB API.

In his PhD thesis, Reicherdt presents the MeMo tool suite capable of importing MATLAB/Simulink models and applying formal verification on the imported representation [115]. Reicherdt uses the *Simulink Library for Java* for the creation of an initial representation of the Simulink model that is further enriched with compile-time information extracted using the external API of MATLAB/Simulink, while at the same time pre-processing the model to include the concept of virtual lines as introduced by Merschen [95]. Imported models are persisted in a PostgreSQL database via the Hibernate framework that was also used in the PhD thesis of Merschen [95].

Giese et al. [62] present a prototypical implementation for guideline checking and model transformation for MATLAB/Simulink models using the Fujaba framework. The authors first specify a metamodel based on the Fujaba metamodel [23] for MATLAB/Simulink models and extend it to wrap the actual model representation of a MATLAB/Simulink model using the MATLAB/Simulink API. The metamodel presented in this paper only includes elements to represent the functionmodel itself, a small subset of blocks and the lines between them. The resulting metamodel and its accompanying Fujaba tool suite is directly integrated in MATLAB/Simulink and can be used to perform various analyses. As the instantiated model only wraps the actual Simulink model, no standalone representation is available with this approach and the wrapped model cannot be changed.

Another tool adapter called MAnTrAS (MATLAB Analysis and Transformation API for Simulink) for MATLAB/Simulink models is presented by Kolassa et al. in [81]. Again a metamodel of MATLAB/Simulink models is created and instantiated by using the API provided by MATLAB/Simulink, which is similar to the approach used in this thesis. The authors also present techniques that transform a Simulink model during the application of a visitor pattern in MATLAB/Simulink. These transformations target the analyzability of the model by converting 1:n connections between blocks into 1:1 connections or allow to flatten the hierarchy of the model. Similar transformations were also proposed by Merschen et al. in [97] and are used by the tool adapter presented in this thesis.

Massif (MATLAB Simulink Integration Framework for Eclipse) [70] is a tool adapter for MATLAB/Simulink that also imports model data via the MATLAB API and is implemented as a plugin for the Eclipse IDE. Massif employs a metamodel based on the Eclipse Modeling Framework (EMF) that defines corresponding entities for each element of the Simulink data model. Massif supports the resolution of library blocks and model references contained within a Simulink model, but can also perform a shallow import, where model and library references are not resolved. As the tool adapter presented in this thesis, the MASSIF framework also supports the export of once imported Simulink models back into its source format.

In [27] Choi et al. present a tool adapter for IBM Rational DOORS. As data stored within the DOORS server repository cannot be directly accessed, the authors propose the use of a server application started from a DXL-script in a DOORS Client started in Batch-Mode (without a user interface) to communicate with external applications. This interface can be used to extract data from the model, which is supplied by the adapter as an instance of a metamodel defined to represent DOORS data. The synchronization of local changes to retrieved model data with the DOORS server repository is also possible. Using a server started by a DXL-Script to fetch/change data from the repository is also described in the DOORS DXL reference manual [122]. Besides this approach, we propose another adapter concept in this thesis that is based on the OLE Automation protocol developed by Microsoft [101].

As seamless tool integration and interoperability is an important aspect in various domains with a diverse tool landscape, several frameworks have been proposed to provide interfaces for data exchange and services across tool-borders. The ModelBus framework presented by Hein et al. [68] was created as part of the EU project Modelware and uses a SOA (Service Oriented Architecture) based approach. In the ModelBus framework, data and functionalities are published via a central repository using URLs and can be accessed in a service-like fashion. Communication between tools is managed via a ModelBus specific interface that needs to be implemented by the tool vendors to conform to the ModelBus approach.

Gleirscher et al. [63] present the ToolNet integration framework, which covers similar aspects concerning artifact integration in the context of model-based software development in the avionic domain. The authors propose the separation of model data from meta-data by introducing four layers for the data addressed by the framework. The tool layer includes not yet integrated data, while the model layer includes explicit model data extracted from a tool via a tool adapter. This data can further be enhanced by meta-information such as associations or annotations in the integration layer and focuses on concerns across tool boundaries. On the last layer, the logical layer, all information regarding integrated artifacts is made available. By explicitly separating meta data from model data via the model and integration layer, a similar concept is applied with regard to entanglement of model and meta-data as in this thesis. The approach is evaluated by integrating block diagrams from SCADE and requirements documentation from IBM Rational DOORS.

The Crystal project [92] focuses on the creation of a tool interoperability specification and the creation of a reference technology platform. It builds on the result of similar predecessor projects, e.g. Cesar and MBAT. The project aims to create the interoperability specification by incorporating already existing specifications and data exchange formats as the OSLC (*Open Services for Lifecycle Collaboration*) [11], the ReqIf (*Requirements Interchange Format*) and the FMI (*Functional Mockup Interface*). The paper presented by the authors focuses on aspects of tool integration with the OSLC, mainly addressing traceability aspects across tool borders. Unlike the approach presented in this thesis, data is not centrally stored in a repository but links across tool borders are created using the concept of linked data by querying available data via the OSLC interface of a tool and creating a link to a particular set of data using its URI (*Unique Resource Identifier*).

3.1.3 Bibliographic Notes

In this chapter, an approach for the incremental integration of model-based software development artifacts is presented. The concept and architecture of the described approach were partially published in [57, 58].

Parts of the tool adapter for MATLAB/Simulink that are described in Section 3.3.1, were developed during three bachelor's theses. The export function for MATLAB/Simulink models has been developed by Julius Nehring-Wirxel [102] and a part of the integrated visualization of MATLAB/Simulink models has been developed by René Rousseau [133]. The tool adapter for IBM Rational DOORS, described in Section 3.3.2, has been developed during the bachelor's thesis of Mirko Kugelmeier [87].

3.2 Metamodel

One problem that arises during the integration of artifacts from different sources is that each software artifact may have different characteristics with respect to its syntax and its internal structure. To describe the fundamental aspects of integration artifacts, a metamodel is needed that defines basic representation of artifacts, their contained artifact elements and the dependencies and relationships between these elements [19]. This further eases the creation of tool adapters and, subsequently, the representation of the artifacts targeted by the tool adapters, as common elements and concepts are defined in the metamodel.

We decided to create the artshop metamodel based on the Eclipse Modeling Framework [145], as it provides a formal representation of all modeled entities and their attributes. It allows multiple inheritance as part of its class structures, which is resolved by the integrated code generator. The metamodel defines abstract representations of artifacts and their elements, highly customizable attributes, that can be attached to artifacts as well as their elements and includes constructs representing annotations/associations on/between artifacts and their elements.

The following sections will describe the general structure of the metamodel and the properties provided to the concrete model representations implemented by the tool adapters of the framework.

3.2.1 Entity Structure

One concern of the metamodel is to define basic representations for artifacts and their elements. Figure 3.2 shows an excerpt of the metamodel in the form of an UML class diagram, focusing on the classes corresponding to these basic representations. The root class of this class diagram is the *TraceableElement* class, which is the base class of all entities in the artshop metamodel. Entities in the metamodel are separated into model

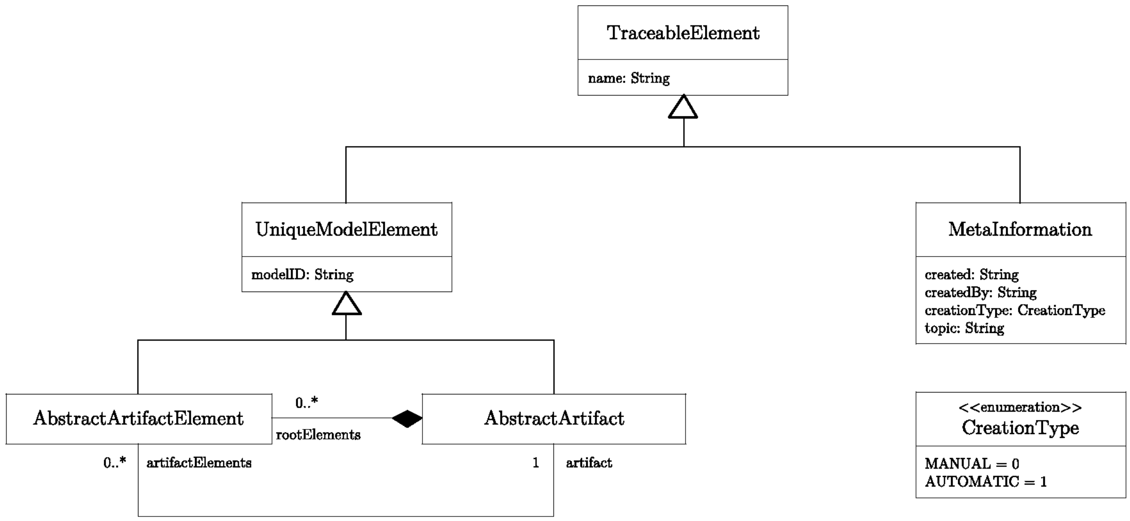


Figure 3.2: Base elements of the artshop metamodel

elements, representing actual development artifacts, and meta-information that represent dependencies or relationships between model elements.

A core property of model elements is their unique id introduced by the class *Unique-ModelElement*. This class is used as the base representation of all model elements. Using the id stored by this class, a model element can be uniquely identified in the context of its defining artifact. This class is further refined into the classes representing artifacts and their contained elements, namely *AbstractArtifact* and *AbstractArtifactElement*. The Artifact class contains bidirectional references to all artifact elements it subsumes and stores a reference to each top-level artifact element. The rest of the internal structure of the artifact has to be defined by refining the AbstractArtifactElement class. These two classes from the metamodel should further be refined to create a concrete model representation based on the metamodel. Examples for the creation of concrete model representations will be given in Section 3.3.

Meta-information is represented by *MetaInformation* instances, which store information about their creation date, author, whether they have been manually or automatically created and a topic by which they can be grouped. Further refinements of the MetaInformation class will be introduced in Section 3.2.3.

3.2.2 Artifact Attributes

A recurring property among model-based software artifacts are dynamic attributes of their contained artifact elements. These attributes typically can be freely defined by the user and therefore cannot be modeled as part of a static concrete model representation. To deal with user-defined attributes, the artshop metamodel introduces the concept of *AttributableElements* as shown in Figure 3.3.

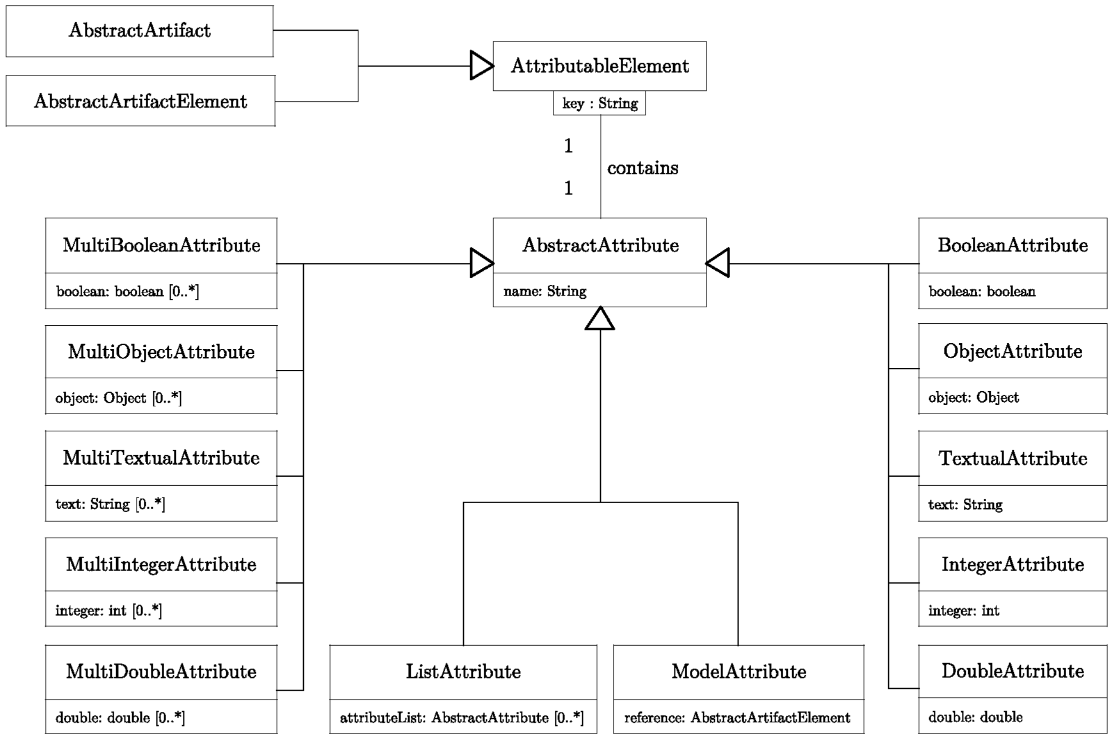


Figure 3.3: Dynamic attributes in the artshop metamodel

The AttributableElement class manages a set of key-value pairs mapping attribute names, also called keys, (typed as Strings) to a corresponding *AbstractAttribute*.

While the AbstractAttribute class only contains the name of an attribute, it is refined by further attributes, which can store values of specific data types. These data types range from single values of primitive data types such as `int`, `double` and `boolean` to more complex data types like `String`, `Object` or references to objects specified by the artshop metamodel.

In addition to the representation of distinct values of specific data types, the metamodel also includes attribute representations that can store multiple values of the same type. The names of these attributes are prefixed with the *Multi* character string and can be seen on the left hand side of Figure 3.3.

ListAttributes allow the hierarchic aggregation of multiple attributes and the representation of complex data structures by nesting available attribute instances in a ListAttribute instance. Further hierarchical attribute layers can be expressed by multiple instances of nested ListAttributes.

By default, both the AbstractArtifact and AbstractArtifactElement class subclass AttributableElement, which enables the use of dynamic attributes across all Artifacts and ArtifactElements.

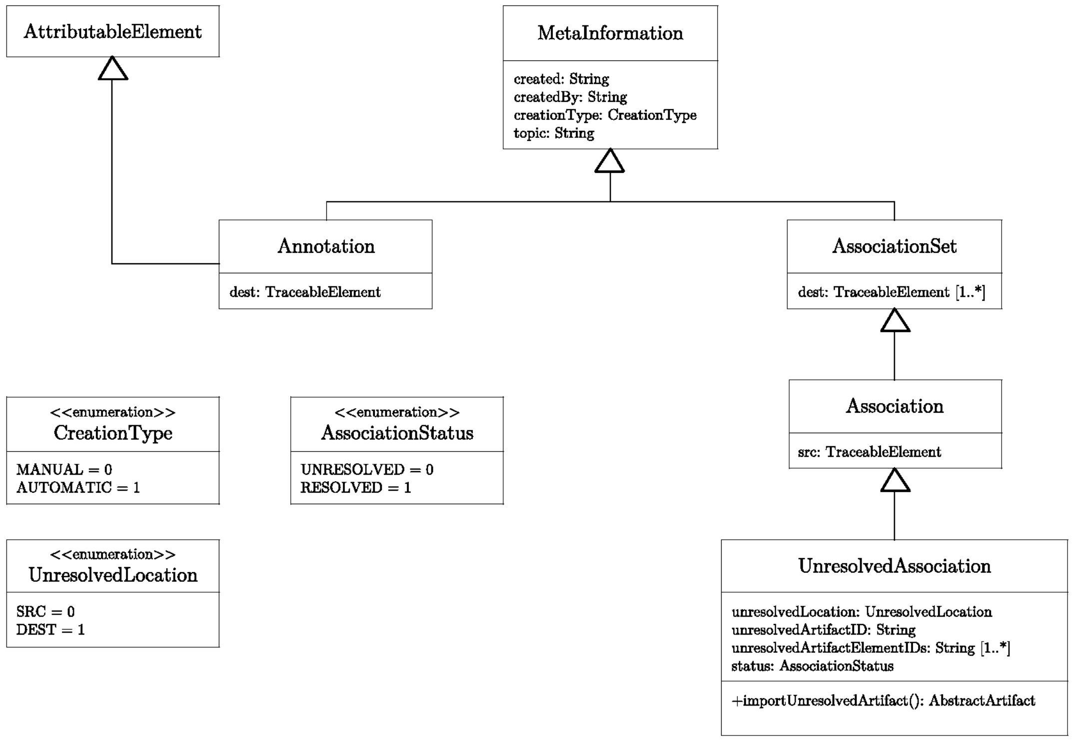


Figure 3.4: Representation of meta-information and associations in the artshop metamodel

3.2.3 Annotations and Associations

Besides the base representations of model elements and their attributes, the artshop metamodel further provides basic concepts for relationships, dependencies and annotations between and to arbitrary elements of the metamodel, called MetaInformation. The data structures of model elements and meta-information are strictly separated by the artshop metamodel to prevent the entanglement of model data with meta information. To realize this separation, we use the asset referencing approach introduced by Schulze et al. in the context of variability management [142]. The idea behind this approach is to use a referencing model that stores the relationships between the elements of one or multiple *assets*, corresponding to artifacts and their contained elements.

The MetaInformation class is further refined by the *Annotation* and *AssociationSet* classes. An annotation represented by the Annotation class can be attached to an arbitrary element. As it also inherits the properties of the AttributableElement class, arbitrary attributes can be attached to an Annotation. This may range from simple textual comments to a set of key performance indicators obtained from static analysis.

While the Annotation class enriches an element with additional information, the AssociationSet class represents a relationship between multiple elements, where each element has the same role. This representation can be used to express an undirected relationship between a set of elements. It is further refined by the *Association* class that adds a source element to the association set. This class can be used to represent directed relationship between one element and a set of elements.

All previously introduced representations of meta-information have in common that their respective destination and source elements have to exist, to be connected by a meta-information. To enable the declaration of an association with one unspecified end, the *UnresolvedAssociation* class is included as a refinement for an association. This class stores essential information that is needed to import the artifact a it points to as well as information needed to identify the actual targets of the unresolved association end within a. The method `importUnresolvedArtifact` needs to be implemented while subclassing the UnresolvedAssociation class using the import operations of its accompanying tool adapter.

3.3 Tool Adapter

To integrate artifact data from a tool in the repository, artifact data has to be processed and converted into a concrete representation based on the artshop metamodel. Depending on the tool the artifact data is created in, artifact data may not be directly accessible from the file system or may even be stored on a server only accessible via the respective client software. To encapsulate tool specific artifact integration processes, the artshop framework uses so-called tool adapters. Tool adapters provide a model based on the artshop metamodel describing a concrete representation of the artifacts of the corresponding tool as well as import procedures to access and convert tool-specific artifacts into this model format. Optionally, tool adapters may include the following components:

- **Export procedures** An export procedure again converts the representation based on the artshop metamodel into its source format
- **Artifact-specific views** To visualize data imported via the tool adapters, artifact specific views can be provided by an adapter for proper visualization in artshop

In the following, we will describe the tool adapters for MATLAB/Simulink, IBM Rational DOORS and pure::variants.

3.3.1 MATLAB Simulink/Stateflow

One of the most extensive tool adapters in the artshop framework is the tool adapter for MATLAB/Simulink. As discussed in Section 3.1.2, there exist three approaches in the literature to realize a tool adapter for the import and export of MATLAB/Simulink models.

- Import/Export by parsing/writing model files [118]
- Import/Export by directly interfacing MATLAB/Simulink via the MATLAB programming language [51, 62, 70, 81, 91, 95]

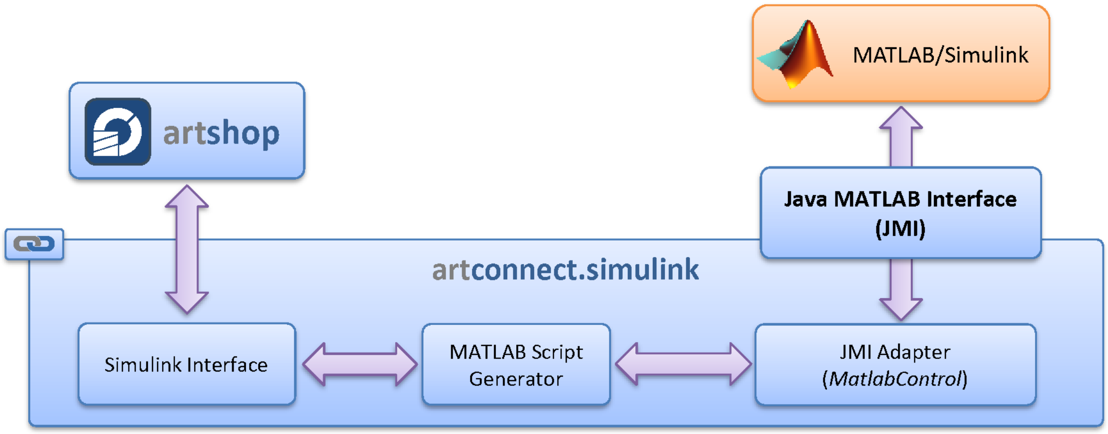


Figure 3.5: Communication between artshop and MATLAB/Simulink

- Hybrid approach of both [115]

While the first approach would be very performant as it only requires parsing the XML structure of the model files, the internal structure of model files from MATLAB/Simulink (*.mdl,*.slx) is neither documented nor stable and may change without notice between releases. The MATLAB API is well documented and can be used to access and manipulate models, but has the disadvantage of requiring a running instance of MATLAB/Simulink. In addition, accessing compile-time parameters, e.g. signal data types, signal dimensions or execution order, of MATLAB/Simulink models is only possible via the MATLAB API. Furthermore, the MATLAB API also eases the resolution of block libraries and model references that can be placed in a MATLAB/Simulink model. A hybrid approach, as implemented by Reicherdt [115], might have advantages regarding the overall performance but still relies on the MATLAB API for the extraction of compile-time information. At the same time, a hybrid approach is still susceptible to changes of the MATLAB/Simulink file format.

The tool adapter presented in this thesis uses the second approach for model import, manipulation and export, as we believe that the stability and well-defined functionality of the MATLAB API outweighs potential performance advantages of the first and third approach. The communication between artshop and MATLAB/Simulink is visualized in Figure 3.5. The Simulink interface of the tool adapter exposes basic model import, export or model manipulation functionality as part of its Java implementation. These functionalities are realized by invoking scripts written in the MATLAB programming language and access or manipulate model data in MATLAB/Simulink. Scripts are supplied by the MATLAB Script Generation component and invoked in MATLAB/Simulink by using the *Java MATLAB Interface*, which allows the execution of MATLAB scripts in a running instance of MATLAB from an application running in or outside of MATLAB itself. To connect the tool adapter to the Java MATLAB Interface we use the *MatlabControl*

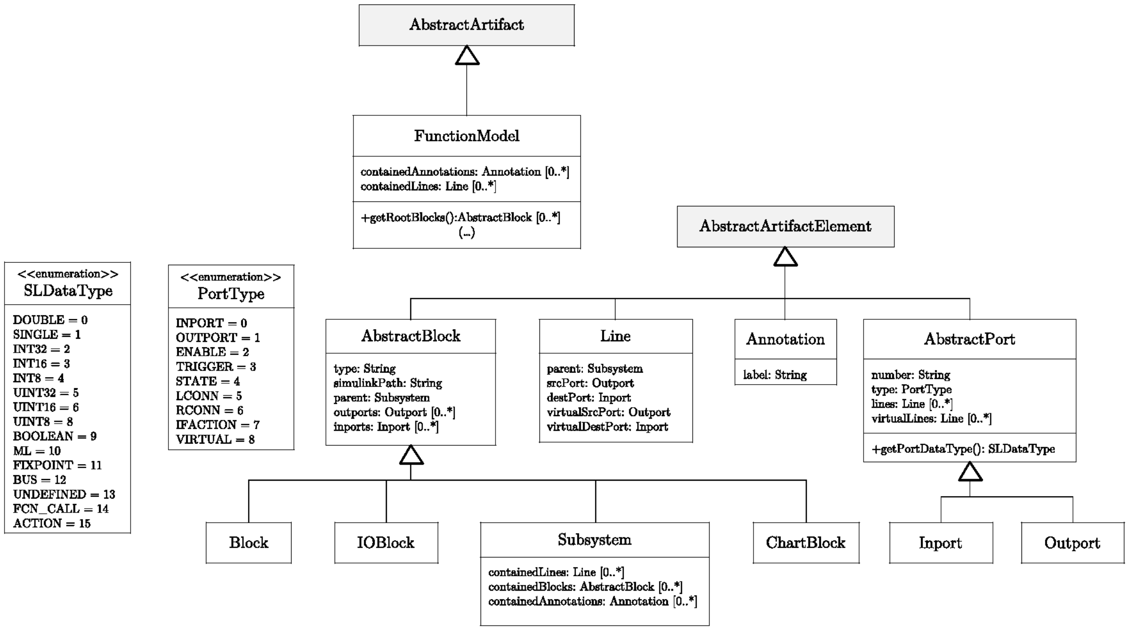


Figure 3.6: Concrete representation of a functionmodel from MATLAB/Simulink based on the artshop metamodel

library[1]. Results of executed scripts are passed back to the tool adapter and can be converted into a concrete model representation based on the artshop metamodel.

Model Representation

A concrete representation of a MATLAB/Simulink model has to include all relevant elements and parameters described in Section 2.2. As each block type has its own set of individual block parameters, it is not feasible to create a concrete model element class for every Simulink block type. Nevertheless, different block types and their corresponding parameters need to be represented accordingly to enable syntactic and semantic analyses of MATLAB/Simulink models.

Figure 3.6 shows the concrete model representation of functionmodels imported from MATLAB/Simulink as a class diagram. Associations between model elements have been omitted to improve the readability of the diagram. Classes with a gray background are defined as part of the metamodel introduced in the previous section. The *FunctionModel* class inherits from the *AbstractArtifact* and represents a functionmodel as defined in MATLAB/Simulink. The top-level elements of a MATLAB/Simulink model are stored within the references defined in the FunctionModel and the AbstractArtifact class respectively. Additionally, the representation defines classes for each type of element present in a MATLAB/Simulink model, e.g. blocks, block ports, lines and annotations. The model distinguishes between four different types of blocks. Each of these classes are

[1] http://code.google.com/p/matlabcontrol/

derived from the *AbstractBlock* class, which contains base properties of each block, e.g. its block type, simulink path, a reference to its containing subsystem or null if the block resides on the top-level of the corresponding model and references to its owned in- and outports. Further block parameters are saved by using the concept of dynamic attributes from the artshop metamodel, as enabled by the *AbstractArtifactElement* class. The *IOBlock* class are used for blocks that define the interface of a functionmodel or subsystem respectively, e.g. *Inport* or *Outport* blocks. The *Subsystem* class is used to represent Subsystem blocks and provides fields to store contained blocks, lines and annotations on its related hierarchy level. Instances of the *ChartBlock* class are used to represent the content of a MATLAB/Stateflow model embedded into the MATLAB/Simulink model. All other blocks not covered by the aforementioned cases are represented by the *Block* class.

Each block has an interface defined by the ports associated to it. Ports are represented by the *AbstractPort* class and store their number, i.e. their position in the interface of the block, their type (as specified by the PortType enumeration) and their associated lines and virtual lines. To distinguish the inports from the outports of a block, the AbstractPort class is refined by two classes representing these ports. Again, port parameters are stored using the concept of dynamic attributes. Each port is typically associated with a datatype that is derived from the signal that is propagated over the port. It can be extracted from its parameters using the *getPortDataType* method.

The *Line* class represents the actual lines of the MATLAB/Simulink model between two ports. As we use the concept of virtual lines introduced by Merschen et al. in [97], lines can further store a virtual connection between two ports connected by indirect signal flow using the *virtualSrc/DestPort* references. Line parameters are again stored using the concept of dynamic attributes as inherited from the *AbstractArtifactElement* class.

Moreover, the *Annotation* class represents textual annotations that can be placed on every layer of a MATLAB/Simulink model.

MATLAB/Stateflow To support the representation of MATLAB/Stateflow models, the ChartBlock class, introduced in the previous paragraph, is refined as shown in the class diagram in Figure 3.7. In contrast to the model representation of MATLAB/Simulink models, all elements of the model are mapped to specific classes. We again use the concept of dynamic attributes to store parameters not directly defined within their respective model classes. The model representation includes elements to represent the *States* of the MATLAB/Stateflow model, including their labels, defined variables and hierarchy levels. *Transitions* describe how the *ConnectableElements*, i.e. states and junctions, of the model are connected with each other and store the corresponding transition label and its execution priority. The *SFVariable* class stores information about the variables of the MATLAB/Stateflow model, which can be used to store intermediate calculation result or exchange data between states. Variables that are used to connect a MATLAB/Stateflow model to its containing MATLAB/Simulink model are represented by the *SFPortVariable* class, which refers to the port its containing value is driven by. User-defined functions are

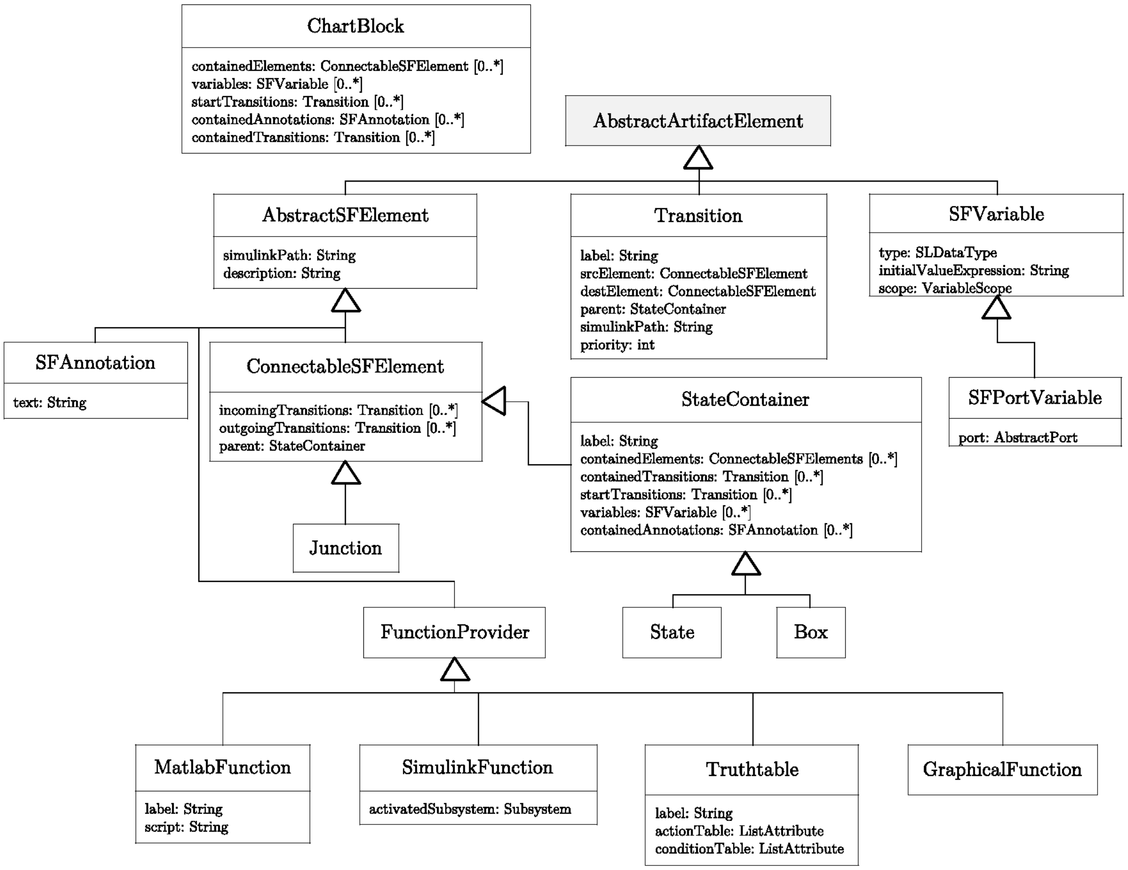


Figure 3.7: Concrete representation of stateflow model elements from MATLAB/Stateflow based on the artshop metamodel

mapped to the specializations of the *FunctionProvider* class, while the *SFAnnotations* class represent user-defined textual annotations on the diagram.

Import Procedure

The tool adapter uses the MATLAB API to access elements of a given model and converts them into instances of the previously shown model representations. Elements in MATLAB/Simulink can be accessed using so-called object handles, which behave similar to pointers as known from traditional computer programming. A handle represent a reference to an actual MATLAB object. After a handle is obtained, the fields of the actual MATLAB objects can be accessed in a generic way by using the built-in `get` function of the MATLAB API. The result of this function is subsequently used to extract the necessary information to instantiate and populate the concrete model representation and instantiate dynamic attributes for each field not directly addressed in the respective model representation class. Elements are imported from a MATLAB/Simulink model in a top-down fashion.

1. **Functionmodel** The model object is loaded and its attributes are imported
2. **Blocks** Blocks contained in the loaded model are imported, including their parameters
3. **Ports** Port handles are extracted from the imported blocks and imported including their parameters
4. **Lines** Line handles are extracted from the imported functionmodel/subsystem blocks and imported including their parameters. Line branches are resolved to individual lines connecting exactly two ports with each other
5. **Annotations** Annotation handles are extracted from the functionmodel and imported
6. **Resolve Stateflow models** Resolve all Stateflow models associated with the ChartBlocks imported during Step 2 by using the MATLAB/Stateflow API in a similar fashion as the import procedure for MATLAB/Simulink elements

As the *FunctionModel* instance is populated by imported elements, relationships between elements, e.g. containment relationships of ports within blocks, of blocks into their corresponding containing element or the assignment of source and destination ports to created lines, are resolved in the created model representation. During the import, all visible parameters of each element are imported and saved as dynamic attributes as specified in the artshop metamodel. Compile-time parameters, e.g. execution order (see Section 2.1), signal dimensions and dimensions, are extracted if the model can be compiled. Otherwise the import of these parameters is skipped.

Model libraries are automatically resolved during step 2 of the import procedure, while model reference blocks can be resolved by activating an option prior to the import procedure.

The result of the import procedure is a coherent instance of the model representation presented in the previous section.

Parameter deduplication Since typically many parameters are shared across blocks, ports and lines, the tool adapter reassigns recurring dynamic attribute values among instances of the model representation elements, reducing the amount of instances of the AbstractAttribute class from the metamodel by up to 99%, without losing information. Figure 3.8 shows the inner structure of the component responsible for parameter deduplication. After all attributes for an AttributableElement e have been created, the attributes of e are entered into the queue of the attribute deduplication component. Attributes are retrieved by the attribute classifier concurrently to the thread that handles the creation of the actual model elements. The classifier classifies an attribute based on the type of its owner and retrieves an attribute processor based on the identified class. This processor is responsible for managing all attributes for a given equivalence class and for the actual deduplication operation. The attribute is then entered into a worklist associated to its assigned processor. After all attributes from the queue have been

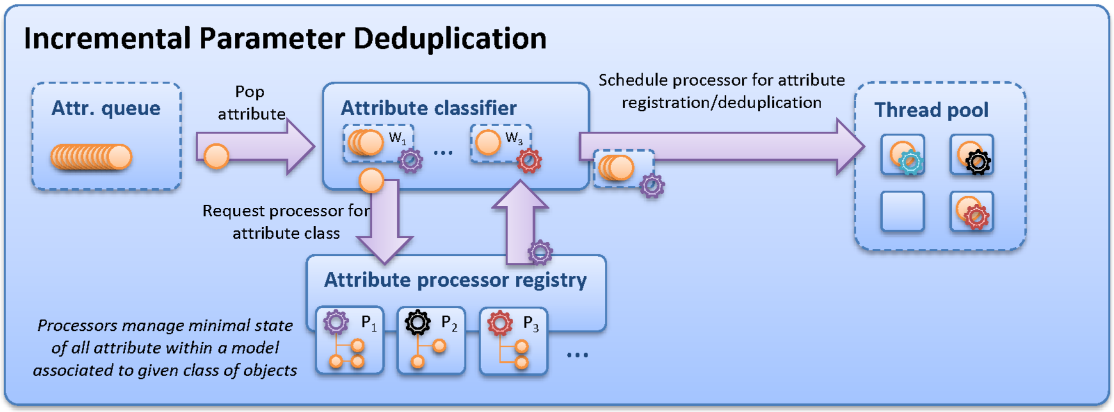


Figure 3.8: Overview of the incremental parameter deduplication component

assigned to a worklist, the processors are scheduled once a slot in the thread pool of the component becomes available. Processor scheduling is prioritized based on the backlog of the worklist associated to the respective processor. Attributes assigned to a processor are again internally classified based on their name, i.e. only attributes with the same name are considered during attribute deduplication. Once scheduled, the processor looks up already processed attributes based on the attribute class of the currently processed attribute and performs a deep-equal comparison against the current attribute. If a match is found, the current attribute is replaced by the matched attribute. If no match is found, the processor puts the current attribute into the registry associated with its attribute class. This process is repeated until the worklist of the processor is empty.

Virtual line creation To ease the navigation on the created model during model analyses, the concept of virtual lines has been adopted from Merschen et al. [97]. The authors enriched each line representation in an imported MATLAB/Simulink model by a virtual source and destination, to enable the navigation of implicit signal flow (see Section 2.2.1) across subsystem boundaries and between Goto and From blocks. During virtual line creation, the virtual destination/source port of each line attached to a subsystem is set to a virtual port created on the In-/Outport blocks corresponding to the actual destination/source port of the line as it can be seen in Figure 3.9. In this figure, the virtual part of each line is shown as a separate dashed line. Virtual ports are marked by setting their `portType` attribute to the enumeration value `PortType.VIRTUAL`. Goto/From blocks that communicate via implicit signal flow are handled in a similar way, but as no line exists between these blocks, a new Line instance is added between the created virtual in- and outports. This concept was extended to respect different configurations of Goto/From blocks regarding their scope (accessibility) in the model, by considering the scope of the Goto block (local, global and scoped) and detecting all From blocks that are enclosed in the given scope. An example for a virtual line connecting a Goto and From block can be seen in the lower half of Figure 3.9.

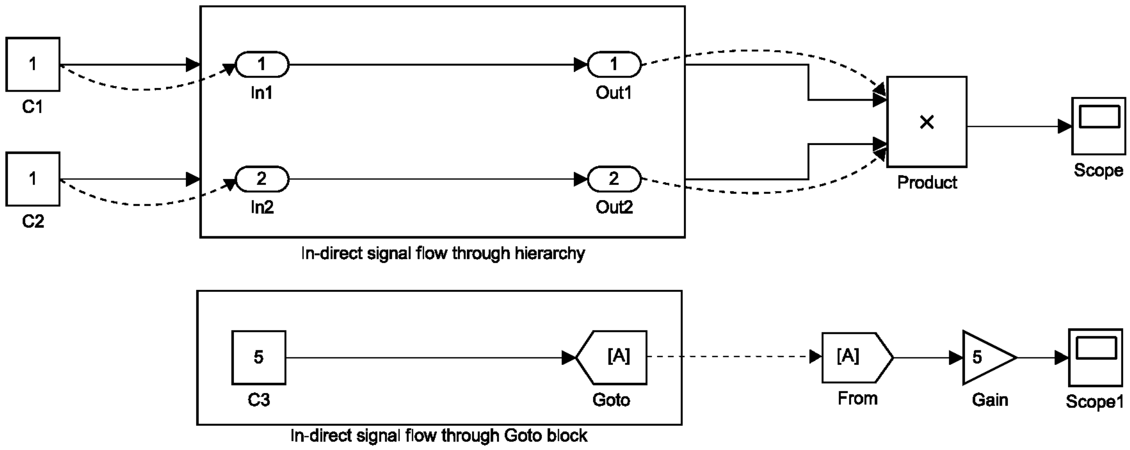


Figure 3.9: Visualization of virtual lines between hierarchy layers and Goto/From blocks of a model

The algorithm further has been extended to connect the active variant in a VariantSubsystem block, which is a special type of subsystem block that contains multiple subsystems corresponding to variants of the functionality implemented by the block. The subsystem corresponding to the active variant, set in the block parameters of the VariantSubsystem block, is connected to the In-/Outport blocks of the VariantSubsystem by line elements. The aforementioned virtual line calculation algorithm is further used to add virtual source and destination ports for the computed virtual lines.

At last, further line elements are added between Inport blocks and their associated ShadowInport blocks, which output the same signal as its associated Inport block. Each ShadowInport block is connected to its associated Inport block by adding a line to a virtual inport created on the ShadowInport block.

Tool Specific Views

A typical problem that arises when trying to visualize analysis results in MATLAB/Simulink is that visual changes to a model element in the tool are directly changing the underlying model representation. Furthermore, the visual transformation capabilities of MATLAB/Simulink are limited to annotations added on the canvas and changing the color of elements. To support the creation of views and the annotation of information to elements based on analysis results or user input, views have been created for the representation of MATLAB/Simulink and MATLAB/Stateflow models, which can be used to visualize, navigate and visually enrich the imported model without changing its underlying representation. Figure 3.10 shows the realized visualization for the top-level of the demo model sldemo_fuelsys from MATLAB/Simulink. Elements in the views are created based on the layout information of the imported model elements. The view supports the annotation, filtering, highlighting and selection of displayed elements. New views can either be created and altered manually or based on the results of analyses.

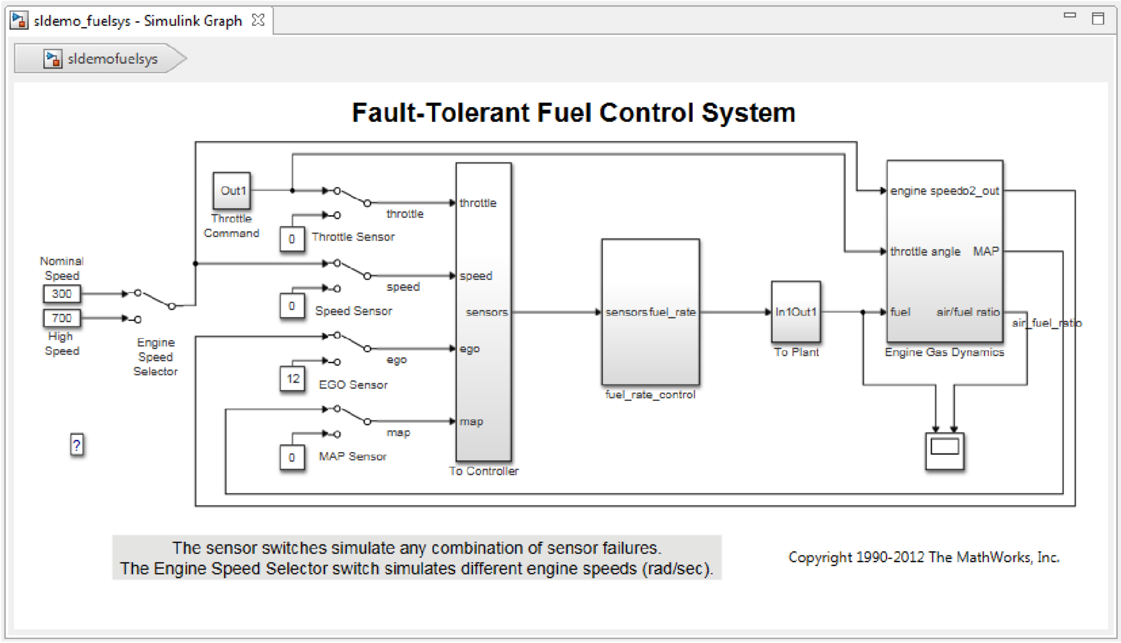


Figure 3.10: Examples for the visualization of imported model elements from MATLAB/Simulink using the demo model sldemo_fuelsys (MATLAB/Simulink 2014a)

Export Procedure

The tool adapter further offers the functionality of exporting an imported model back to MATLAB/Simulink. This functionality was developed as part of the bachelor's thesis of Julian Nehring-Wirxel [102]. Currently, the export function is limited to the export of elements from MATLAB/Simulink but could be extended to support MATLAB/Stateflow models. The export functionality also utilizes the JMI interface introduced at the beginning of Section 3.3.1 to create model files and elements. The workflow of the export functionality is organized in a top-down fashion.

1. **Create model:** First, a new model file is created that will hold the exported model and loaded to set the parameters of the actual model element
2. **Create blocks:** In this step, all blocks are placed in the newly created model file and their corresponding block parameters are set to the attributes present in the artshop model representation
3. **Create lines:** After all blocks have been created, all lines have valid source and destination blocks and can be created and enriched with their corresponding parameters
4. **Create annotations:** Finally, the annotations of the model are created

One important characteristic of model elements from MATLAB/Simulink is the unique id (SID) that is attached to each block of a model. As these ids depend on the actual

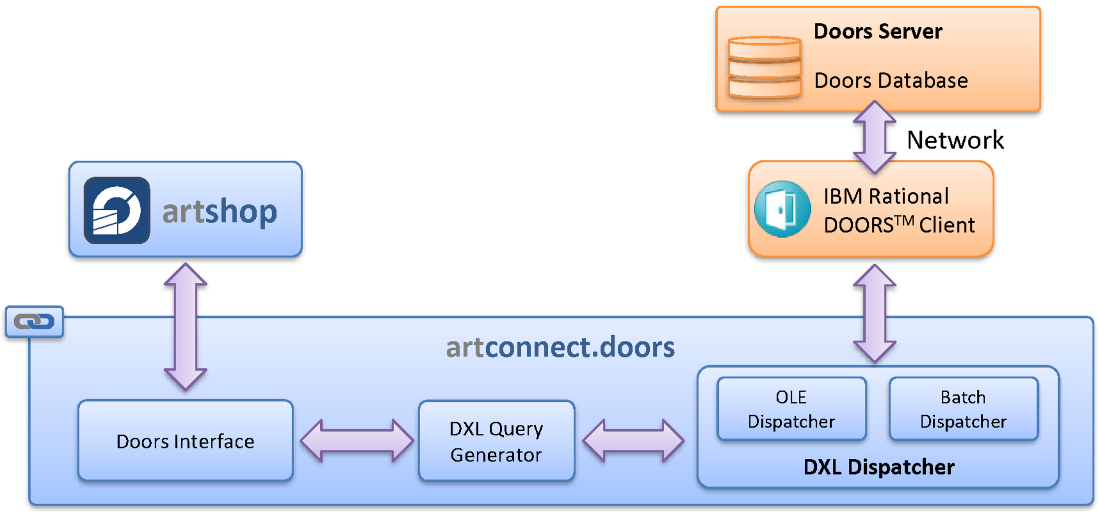


Figure 3.11: Communication between artshop and IBM Rational DOORS

order of creation of elements in a model, these ids would not match the ids that were present in the initial artshop model representation. Unfortunately, there exist no function in the MATLAB API to manipulate the SID of a model element. To fix the SIDs after the export has finished, the XML representation of the created model file is traversed after the export is finished and SIDs are set to the values present in the artshop model representation.

3.3.2 IBM Rational DOORS

IBM Rational DOORS was introduced as a client-server application, which stores its data on the server-side to enable collaborative use of authenticated clients in Section 2.4.2. The development of a tool adapter for IBM Rational DOORS therefore has to leverage a trusted communication channel to the DOORS server, i.e. the client application. By using the DXL scripting language, data can be queried and modified on the server by executing DXL scripts via an authenticated client application. The DOORS DXL reference manual [122] describes two approaches to realize data exchange with the DOORS client and an external application.

- Start the DOORS client in batch mode and execute a script acting as a DXL execution server by opening a network port to receive and execute DXL scripts against the data managed by the DOORS server. Starting a DOORS client in batch mode requires user credentials during the authentication process against the DOORS server. This method was also used in the work of Choi et al. in [27]

- Execute DXL scripts in a running DOORS client application by transferring scripts via the OLE Automation protocol [101]. This approach is only available on computers with the Windows operating system

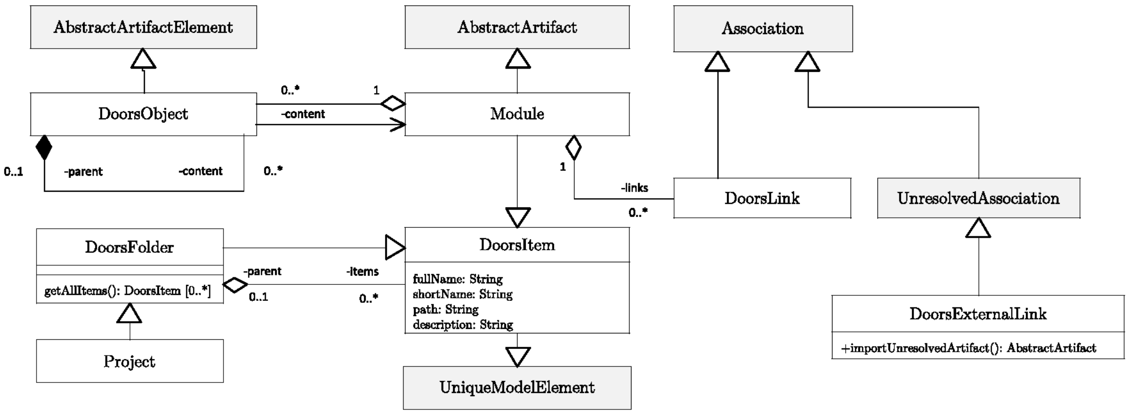


Figure 3.12: Concrete representation of elements from IBM Rational DOORS based on the artshop metamodel

As shown in Figure 3.11, the tool adapter implemented as part of this thesis can use both of these approaches to exchange data with a DOORS server. The tool adapter exposes an interface, which provides basic functionality such as model import and export as part of its Java implementation. The interface delegates these functions to the DXL query generator component, which translates requests to corresponding scripts in the DXL scripting language. Generated scripts are executed via the DXL Dispatcher component by either using the OLE automation protocol or by starting a DOORS client in batch mode. Results of the executed DXL scripts are encoded in the JSON (JavaScript Object Notation) format, a compact text-based data format, and passed back to the tool adapter. The tool adapter converts received JSON objects into instances of the DOORS model representation, which are returned to the application that requested them.

Model Representation

The tool adapter targets formal modules saved on a DOORS server that represent documents storing development specific information, e.g. requirements or test specifications. As mentioned in Section 2.4.2, formal modules are organized in folders and projects that can be nested into each other. To represent the organization hierarchy a formal module resides in, the model representation of the tool adapter has to contain classes for projects and folders, beside the class of formal modules. Figure 3.12 shows the classes used by the tool adapter to represent the aforementioned elements. The *DoorsFolder* and *Project* classes are only used to represent the organization hierarchy of the server and are not intended to be saved as a result of the import procedure. Formal modules are represented by the *Module* class and contain a set of *DoorsObject* elements that represent the rows of the module. To represent the hierarchical structure of a Module, DoorsObjects may again contain child objects. The concept of dynamic attributes is used to represent the columns of a DoorsObject, as the columns are customizable by the user. All model elements of IBM Rational DOORS can further be identified by a unique id that is stored in the attribute inherited by the UniqueModelElement class provided by the metamodel.

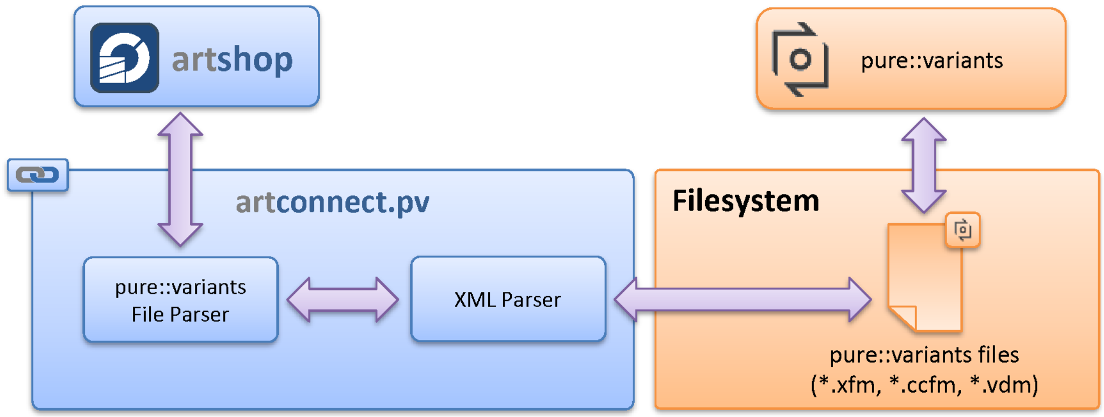


Figure 3.13: Communication between artshop and pure::variants

As DoorsObjects in a formal module can be linked to other DoorsObjects, either within the same or another module on the DOORS server, classes representing these traceability links need to be provided. The *DoorsLink* class, that inherits the base functionality of the Association class, represents a link in the same Module. To represent links to other modules the *DoorsExternalLink* class is used, which inherits the functionality of the UnresolvedAssociation class to link to an element that is not available in the scope of the currently imported Module. Additionally, the ExternalDoorsLink class implements the needed methods to import/resolve the Module its unresolved association end resides in. Once resolved, both links are associated with their defining Module element.

Import and Export Procedure

The import of formal modules from IBM Rational DOORS is realized using DXL scripts by first importing the hierarchy and the formal modules of the DOORS database excluding its content. After converting received data from the query into an instance of the representation introduced in the previous section, a formal module can be selected and imported by the adapter. The import procedure of a module is organized as follows:

1. **Import Module metadata** In the first step, metadata of the module is imported. This includes attributes like the module description, last modification/modifying date/user, its creation date and the user that created the module.

2. **Import DoorsObjects** The hierarchy of the module is then extracted and all containing DoorsObjects are created. This includes metadata and all attributes representing the rows of the created DoorsObjects.

3. **Import Links** Finally, all links of the module are imported and linked to their respective elements. ExternalDoorsLinks are created for links that target a currently unresolvable model element.

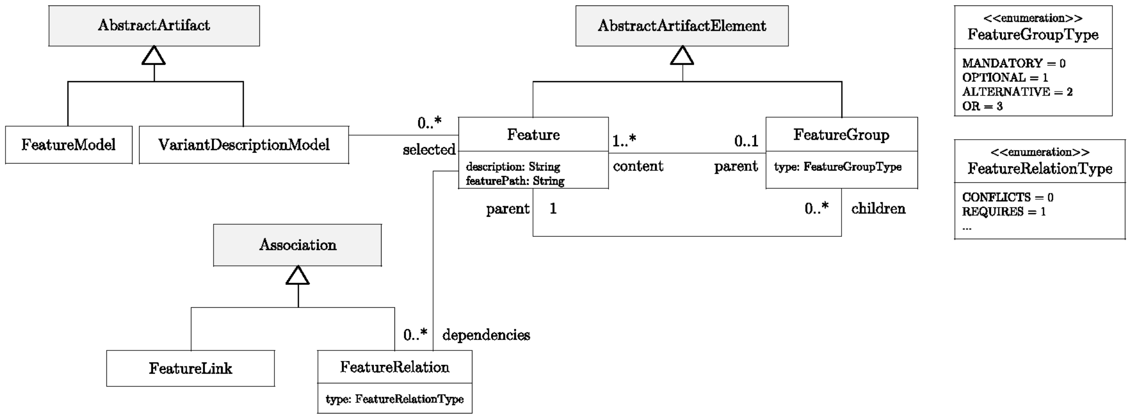


Figure 3.14: Concrete representation of feature and variant description model elements from pure::variants based on the artshop metamodel

Once imported, a Module can be exported to a DOORS server by importing the hierarchy of the DOORS database and selecting a folder/project within the returned representation where the module should be saved in. After that, the artshop representation of the module is used to create a DXL query that creates a new module at the chosen location and recreates the module's structure and content in the DOORS database.

3.3.3 pure::variants

The tool pure::variants, introduced in Section 2.4.2, is a desktop application used to manage variability documentation in the form of feature, family and variant description models. Models are saved as XML documents, which contain all information present in the tool.

The tool adapter for pure::variants directly imports model data from the model files created in the file system as shown in Figure 3.13. The import process is started by handing a path to a pure::variants model file to the pure::variants model file parser that then parses the file via a standard XML parser and instantiates a model corresponding to the content of the parsed model file.

Model Representation

The tool adapter targets three different kinds of models created as part of the variability documentation of a software product line: feature, family and variant description models. Figure 3.14 contains the metamodels used for feature and variant description models. Latter only contains a list of features that are selected to form a concrete variant out of the features contained in a linked feature model and is represented by the *VariantDescriptionModel* class. *Features* are contained in the *FeatureModel* class as part of the content and rootElement references introduced by the AbstractArtifact class. *FeatureGroups* again contain feature elements and are associated with a FeatureGroup enumeration value that expresses the category of features contained within the feature

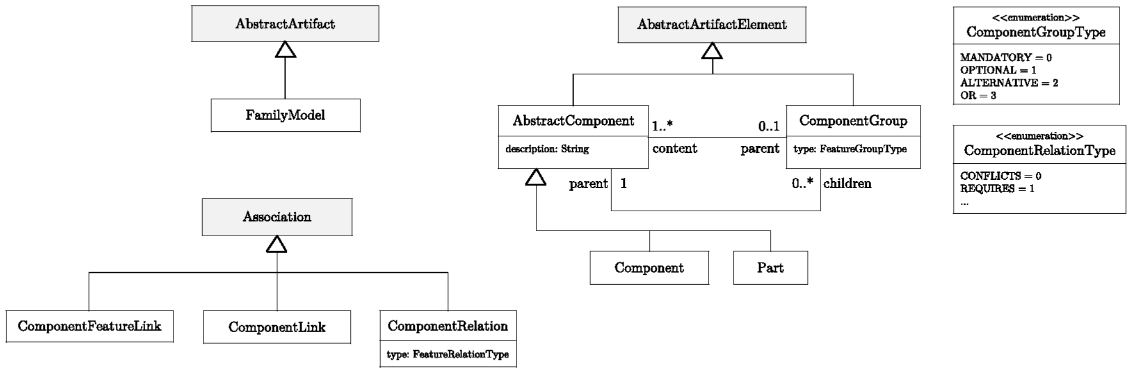


Figure 3.15: Concrete representation of family model elements from pure::variants based on the artshop metamodel

group. This enables the distinction of mandatory, optional and alternative features as introduced in Section 2.3.2, as well as further feature categories that are application specific to pure::variants and are omitted within the diagram. To express feature interdependencies as requires and exclude relationships, the *FeatureRelation* class has been created. It is derived from the association class of the artshop metamodel that already provides means to link multiple artifact elements with each other. The feature relation is typed by the *FeatureRelationType* enumeration, which contains further pure::variants specific dependency types. To link arbitrary artifact elements with a given feature, the *FeatureLink* association can be used to allow the creation of custom feature associations that allow artifact elements of other tool adapters to be linked to a feature.

Besides feature and variant description models, the tool adapter also provides means to represent family models. An overview of the classes used to represent family models is given in Figure 3.15. The overall structure of the family model in pure::variants is very similar to the structure of feature models. Variability instantiations are represented by the *AbstractComponent* class that is further refined by the *Component* class representing a partition of the solution space that realizes the particular variability component. Atomic partitions of the associated artifact are represented by the *Part* class. With features being categorized by feature groups, the *ComponentGroup* class is used to categorize components by the types provided by the *ComponentGroupType*. Moreover, interdependencies between Components are expressed by the *ComponentRelation* and are categorized by the *ComponentRelationType* enumeration. Dependencies between feature model and family model are represented by *ComponentFeatureLink* associations, while *ComponentLink* associations can be used to connect components to their actual representations in the solution space that were imported by other tool adapters.

Import Procedure

Figure 3.13 shows that model data created with pure::variants is imported by parsing the XML files created by the modeling tool. As the XML format of the three model files is supported by the tool adapter, a generic parser has been implemented that

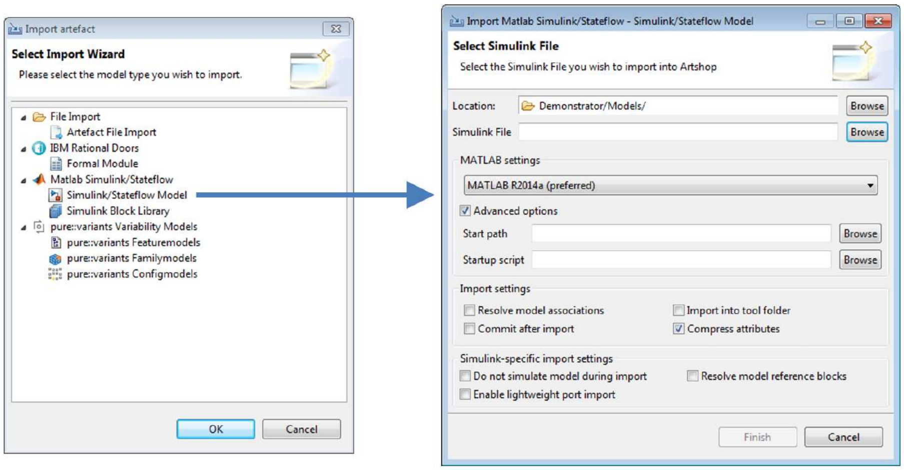


Figure 3.16: Import wizards in artshop

converts the content of the respective XML files into an intermediate representation. This representation is then used to instantiate the EMF models presented in the previous section by model specific converters. Created model files are then passed back to the calling application. As variant description and feature models contain a reference to a corresponding feature model, the importer requires an imported feature model instance as a parameter to resolve cross-references between these models.

3.3.4 Tool Adapter Integration in artshop

Tool adapters are integrated as plugins in the artshop framework. Once registered, the import procedures of the adapters can be selected by the user in the import wizard selection dialog shown on the left-hand side of Figure 3.16. Here, all available tool adapters and their import procedures are listed. An import wizard can only be selected if its prerequisites are satisfied, e.g. an installation of MATLAB/Simulink has been detected on the client computer to activate the tool adapter for MATLAB/Simulink. Otherwise, an error is displayed and the adapter is deactivated.

The right-hand side of Figure 3.16 shows the import dialog of the MATLAB/Simulink tool adapter. Besides the selection of the model file that shall be imported into the tool, the version of MATLAB/Simulink that shall be used for the import procedure can be selected. Available versions are automatically detected on the client system. Furthermore, a start-up script can be selected that is executed prior to the start of the import procedure. Other options allow the deactivation of the parameter deduplication procedure, the direct commit of imported model files into the model repository or the activation of the model reference resolution procedure.

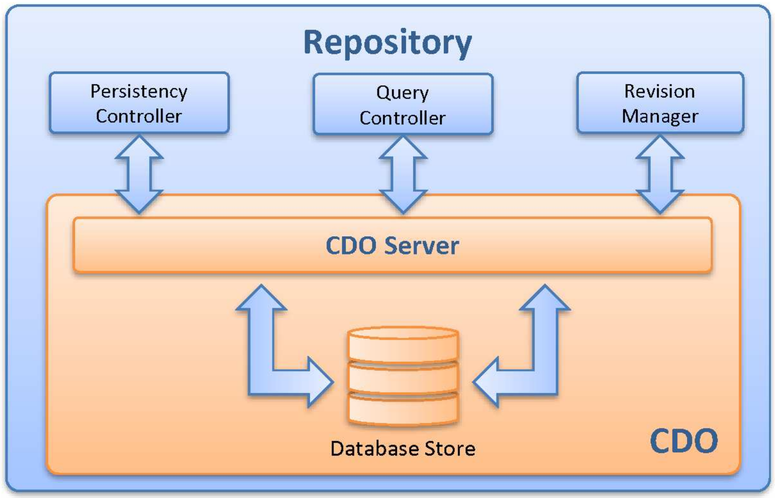


Figure 3.17: Detailed view of the repository component

3.4 Repository and Synchronization

While the tool adapters presented in the previous section provide means to import and export model data from external applications, this section describes the technologies used in the artshop framework to persist and update imported models. After describing the technologies used related to the model repository, the model synchronization mechanism of the artshop framework is introduced.

3.4.1 Repository

The repository component of the artshop framework manages the storage and retrieval of model elements into/from the repository backend. It supplies the basic functionality of a version control system for models imported via the tool adapters. Figure 3.17 shows the innards of the repository component. The main component of the repository is the backend based on the CDO model repository framework [119].

The CDO model repository is a distributed shared model framework for EMF models supporting, among others, the persistence of models in a database, multi-user access/-collaboration, lazy model loading strategies and auditing of past model versions. CDO uses so-called stores to abstract the database technology realizing the actual storage of data, while supporting various relational, non-relational and object databases from different vendors. It is possible to query the models stored in the database to search for specific model elements and/or properties. Supported query languages depend on the used store and the database type it supports. The artshop framework configures CDO to use a relational store (DB) with an in-memory SQL database (H2), and supports both SQL and OCL to query the underlying model repository. The schema of the underlying database is automatically derived from the definition of the EMF model when a model is registered in the repository. Objects committed to the repository are then mapped to rows within tables that were created corresponding to their respective classes. This is different to the approach used by Merschen [95], which required an a priori design of a relational database schema to map created objects to tables in a database. The repositories used in the MATE [91, 148] and MESA [51] projects were derived from the underlying metamodels and therefore all metamodels need to be known a priori and cannot be added dynamically. When using CDO, new model representations can be added to the repository in an ad-hoc fashion, which might happen when a new tool adapter is added to the framework or new model representations are added to the database. CDO can also be configured to track changes of model elements imported into the repository. Past model revisions can then be accessed via read-only views.

The functionality of the CDO framework can be accessed in the artshop framework via three distinct components. The persistency controller handles the storage and commit of unversioned or changed EMF model data, while the query controller provides generic DAOs (Data Access Objects) to query objects from the repository, as well as the possibility to execute queries against the actual model repository. Revision information can be retrieved via the revision manager to obtain details about the changes between different model revisions.

3.4.2 Synchronization

As artifacts are frequently changing during software development, importing an artifact through a tool adapter only provides a snapshot of the artifact at the time of import. The CDO framework can already track changes between different model revisions; therefore, a synchronization mechanism is needed to incrementally update the model data available in the repository to the most current version of the respective artifact. By using an incremental approach, changes made to elements already present in the database can be mapped to these elements and tracked by the CDO framework, while at the same time preserving references to this element from other sources, e.g. meta information such as associations or annotations.

IBM Rational DOORS already includes change management as part of its internal management system and stores change information between user-defined baselines. The change management systems of individual tools as well as their stored information could potentially be used as part of the synchronization mechanism, but as not all tools do supply such fine-grained change information, a more generic solution is required. The synchronization mechanism should be applicable to arbitrary instances of model elements derived from the artshop metamodel, to avoid the implementation of custom synchronization algorithms for each model representation supplied by the tool adapters. Due to the explicit distinction between model data and meta-information, the synchronization algorithm only needs to calculate changes in the model data already present in the repository with regard to an updated version of the same artifact. Therefore, a mechanism is needed to identify and merge changes on instances of arbitrary models derived from the artshop metamodel. After such a mechanism is applied to an existing model instance, the information hold by the synchronized model instance must be equal to the updated source artifact.

One reason EMF was chosen as the underlying modeling technology for the model representations of the artshop framework is that the structural features of its model element instances are easily accessible at runtime, i.e. all fields and attributes of EMF model instances can be accessed in a generic way.

The EMF Compare framework [21, 152] leverages this property of EMF models to implement a generic model comparison and merging algorithm. The comparison algorithm implemented by EMF Compare takes two loaded EMF models, m_1 and m_2, and successively runs through the following phases:

- **Matching:** In the matching phase, the framework tries to map elements from m_1 to elements from m_2. By default, this is done via identifiers if the objects have one or by the use of a distance mechanism. This behavior can be customized to use a custom matching function. If no match is found for an object this is registered and handled accordingly in the next phase. The result of this phase is a set of matches containing either a matched pair of objects or an unmatched object from one of the models m_1 or m_2.

- **Differencing:** In the differencing phase, the differences of all matches are determined using the aforementioned general access to the objects structure. Again, the differencing phase can be customized to, for example, ignore certain structural features. Differences are created for changes in structural features, e.g. the name attribute of an AbstractArtifactElement changed from 'Name' to 'New Name', or for added/deleted objects. The latter are typically created for new objects that could not be mapped to another object in the matching phase.

- **Equivalences:** In some cases, two distinct differences calculated in the differencing phase might actually represent the same change, therefore all differences are compared to each other to link equivalent differences together.

- **Requirements:** To be able to merge differences into either m_1 or m_2, the framework needs to determine dependence relations between differences, i.e. a difference is dependent on another difference if it cannot be merged without it.

- **Conflicts:** In case m_1 and m_2 have both been changed, the framework determines conflicts between the previously detected differences. These conflicts need to be resolved during the merge process.

Differences detected by EMFCompare can be merged from m_1 to m_2 or vice versa on an individual basis. All phases of the comparison and merge process can be customized or adapted to the actual use-case as it has already been noted for some of the phases of the comparison process.

As all models in artshop are represented by EMF models, it is possible to use EMF Compare to realize the aforementioned model synchronization mechanism. While alternatives to EMFCompare exist, as mentioned in a survey of model differencing techniques by Kolovos et al. [83], the seamless integration of EMFCompare in the EMF ecosystem eases the integration of the algorithm into the framework. To realize the synchronization

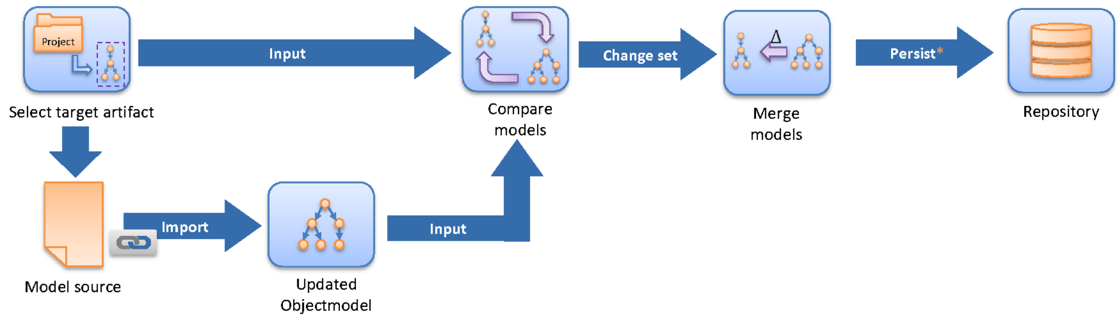


Figure 3.18: Procedure of the artshop synchronization mechanism

mechanism in a generic way, the updated artifact is again imported using the import procedure of the corresponding tool adapter and compared with the artifact already present in the repository using a customized version of the EMF Compare framework as shown in Figure 3.18.

To ensure the correct matching of elements during the matching phase, the unique ids provided by the UniqueModelElement class of the artshop metamodel are used in a custom match function. A subset of structural features related to the concept of dynamic attributes has been excluded during the differencing phase, because of performance reasons due to large amount of soft references to shared attributes in the model, which needed to be processed. As certain assumptions can be made on the structure of these references, a custom merge procedure for these structural features is provided, which significantly increased the performance of the merge procedure. This includes specific pre- and post-processing routines that are applied before and after the execution of the merge procedure of EMF Compare. Within the pre-processing routine, a mapping for the shared attributes of m_2 is created, that maps all attributes from m_2 to the ids of their corresponding AttributableElements. This mapping is used during post-processing to merge all attributes into the merged model m_1 by mapping the Additionally, the customized version of EMF Compare changes the behavior of the actual difference detection algorithm, as the default behavior does not guarantee to preserve the ordering and completeness in multi-valued attributes, once differences on this kind of structural features are merged.

Committing the merged model creates a new revision of the affected artifact in the repository while its previous versions can still be accessed.

Change information of a given revision with respect to its previous one can be accessed through the GUI of the artshop framework. Figure 3.19 shows the history view and the change information for a block of a MATLAB/Simulink model. In the bottom history view, all revisions including the committer and timestamp can be seen, while the upper-left view shows in-depth change information of the elements contained in the selected Subsystem block.

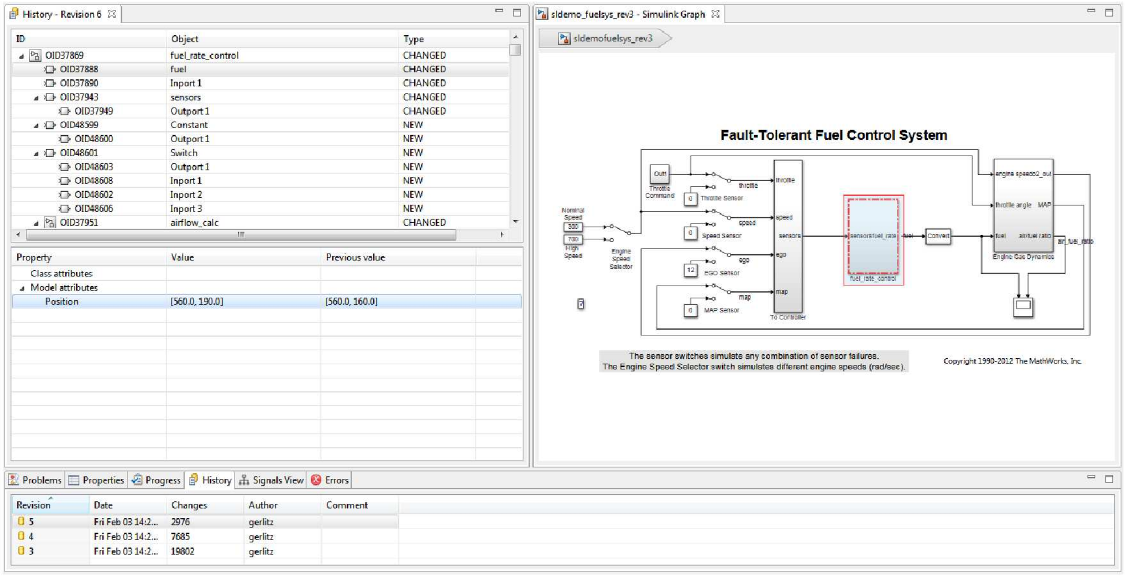


Figure 3.19: Inspection of history information for a block in artshop

3.5 Evaluation

Throughout this section, we will present the results of the performance evaluation of the presented tool adapters, repository and synchronization mechanism. The evaluation has been performed on a desktop computer equipped with an Intel Core i5-6600 processor with two cores, which run at 3.3 GHz, 16 GB RAM and a solid-state drive with 256 GB of memory. The operating system of this computer is Windows 7. During the evaluation, no other processes beside the system processes were running.

First, the evaluation of the tool adapters for MATLAB/Simulink and IBM Rational DOORS is presented. Second, the performance evaluation of the repository and the synchronization mechanism are shown.

3.5.1 Evaluation of the MATLAB/Simulink Tool Adapter

As part of the evaluation, the tool adapter for MATLAB/Simulink presented in Section 3.3.1 is evaluated against the tool adapter presented by Merschen [95] and version 0.5 of the Massif framework [70]. The evaluation includes a short discussion about the model data from MATLAB/Simulink considered by the import processes of the evaluated approaches before the performance of the import and export processes is evaluated.

The major difference between the approaches proposed by Merschen and the artshop tool adapter/Massif, is that latter are executed outside of MATLAB itself, while the import procedure of Merschen is realized as a MATLAB script executed within the tool itself. All approaches use the MATLAB API to extract model data from a loaded Simulink model. Table 3.1 gives an overview of the functionality supported by the evaluated tool adapters with respect to the data imported by their import procedures. Another major

Table 3.1: Comparison of supported functionalities of evaluated tool adapters

Functionality	artshop tool adapter	Merschen [95]	Massif [70]
Model parameter	x	-	-
Block elements/parameter/SID	x/x/x	x/-/x	x/partial/-
Port elements/parameter	x/x	-/-	x/-
Line elements/parameter	x/x	x/-	x/-
Stateflow elements/parameter	x/x	-/-	-/-
Compile-time parameter	x	-	-
Library block resolution	always	always	optional
Model reference resolution	optional	-	optional
Parameter deduplication	x	-	-
Export functionality	x	-	x

difference between the artshop tool adapter in comparison to the other adapters is the extent to which parameters are imported. While artshop supports the import of all parameters from all objects considered during the import, the other tool adapters either ignore or only partially support the import of parameters from a Simulink model. The Massif adapter only imports the parameters of each block that are tunable via the user interface of MATLAB/Simulink, also called *Dialog parameters*. In addition, imported parameters are deduplicated by the artshop tool adapter, reducing the overall amount of parameters present in the resulting model instance. Furthermore, the import of statecharts from MATLAB/Stateflow is only supported by the artshop tool adapter. Considering library block resolution, both the artshop adapter and the adapter of Merschen always resolve library blocks within a model, while the Massif adapter offers an import option to configure if they should be resolved or not. Model reference blocks can only be resolved by the artshop and the Massif adapter if configured to do so. Moreover, exporting an imported model to MATLAB/Simulink is also only supported by the artshop and Massif tool adapter. To evaluate the tool adapters, we use them to import a MATLAB/Simulink model containing the control model of a micro aerial vehicle (MAV) that was described by Meyer et al. in [98]. First, the extent of imported model data of the individual adapters is compared to each other. We use a running instance of MATLAB/Simulink 2014b for all tool adapters and assign each adapter the maximum amount of RAM available on the test system.

Table 3.2 shows the amount of imported model elements for each tool adapter. When comparing the amount of imported model elements, all tool adapters import the same amount of blocks and lines. Only the tool adapter by Merschen creates four additional virtual lines representing connections between Goto and From blocks. As mentioned in Section 3.3.1, the artshop tool adapter represents virtual lines in a slightly different format and does not need to create additional line objects. Ports of blocks are only imported by the artshop and Massif adapter. The amount of elements created by the artshop tool adapter differs slightly from the Massif adapter, as virtual ports are created

Table 3.2: Comparison of imported model data on MAV model

Element type	artshop tool adapter	Merschen [95]	Massif [70]
Model parameter	700	0	0
Blocks	1023	1023	1023
Block parameters (with deduplic.)	146850 (13526)	0 (-)	16064 (-)
Ports (including virtual ports)	2093 (2487)	0 (-)	2093 (-)
Port parameters (with deduplic.)	96278 (3975)	0 (-)	0 (-)
Lines	1064	1069	1064
Line parameters (with deduplic.)	37240 (1818)	0 (-)	0 (-)
Total model elements	29030	2193	25698

during the import procedure to create valid source and destination ports for virtual lines as introduced in Section 3.3.1.

As the artshop adapter is the only adapter capable of importing all parameters related to the model itself and its containing elements, only the model imported by our adapter includes an extensive amount of properties. In total 193718 parameters are imported by the artshop tool adapter, which are reduced to 19319 distinct parameters by applying parameter deduplication during their import. On average, the artshop tool adapter imports 143 block parameters, 38 port parameters and 35 line parameters for the elements of the MAV model. Only the Massif adapter can import a subset of the set of block parameters, while not supporting deduplication of duplicate parameters.

When comparing the total amount of elements contained in the imported models, the model created by the artshop tool adapter contains the biggest amount of elements due to the amount of parameters imported, followed by the model imported by Massif framework, which would contain less attributes if the import procedure would also support parameter deduplication. One thing to note is that the model created by Massif creates an identifier object containing the name and simulink path of each line, port and block element imported, which does not appear in the statistics. This information is directly stored in the model elements created by the tool adapters of artshop and Merschen. The model created by the tool adapter presented by Merschen is the smallest one, as neither ports nor parameters are imported.

To compare the import performance of the three adapters, we measured the time to import the aforementioned MAV model for each individual tool adapter. During this performance evaluation, the MAV model was multiplied by the factors 1, 2, 3, 4, 5, 10, 20, ..., 100 to get an impression on how the performance of the individual tool adapters scales with respect to the size of an imported model.

Figure 3.20 shows the import time for each tool adapter for each version of the scaled MAV model. First thing to note about the import performance of the Massif tool adapter is that no performance information is displayed beyond the scaled model size of 10000 blocks because importing the MAV model scaled by factor 20 resulted in an

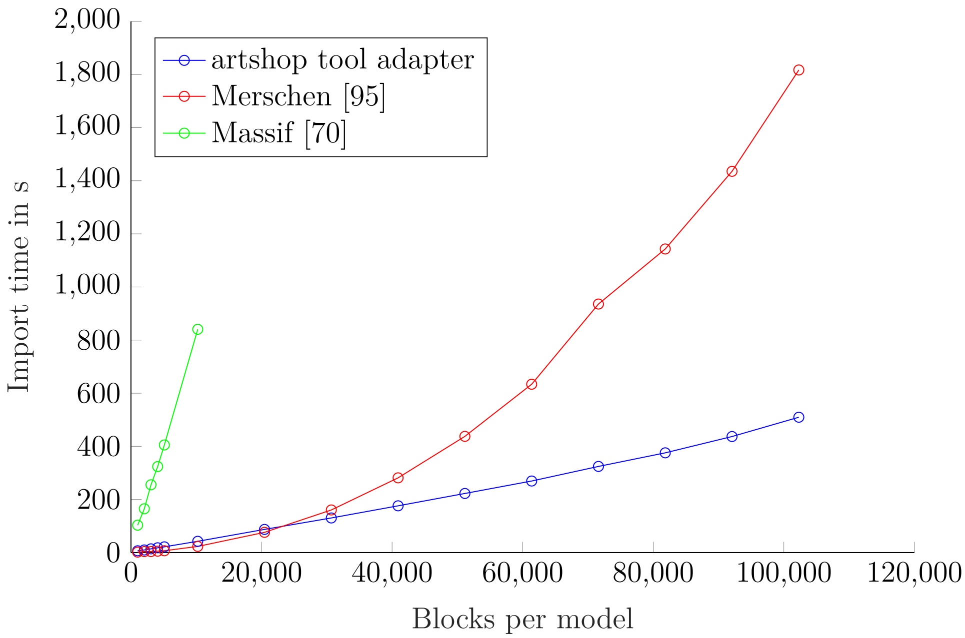


Figure 3.20: MATLAB/Simulink tool adapter import performance for functionmodels containing up to 100.000 blocks

`OutOfHeapException` on our test system, i.e. the import required more heap memory than what was available. In addition, the Massif adapter performs and scales the worst of all adapters, taking 103 seconds for the initial version of the MAV model and 840 seconds for the version scaled by the factor ten. As industrial size models easily reach sizes of around 20.000 - 50.000 blocks or can even be bigger, tool adapters need to reliably work on models of this size, which was only accomplished by the artshop tool adapter and the tool adapter of Merschen. When comparing the artshop tool adapter against the adapter of Merschen, it can be observed that the latter performs better for models containing up to 20.000 blocks, while the artshop adapter performs much better beyond that region. While the curve connecting the import times of Merschen represents an exponential function, the curve of the artshop tool adapter progresses nearly linearly. This is probably related to the limitations of the MATLAB scripting language, leading to performance degradations due to memory consumption, as both the model representation in MATLAB/Simulink and the intermediate data structures of the tool adapter use a lot of memory. The difference between the approaches is at most 10 seconds in the region of up to 20.000 blocks.

Figure 3.21 shows the same data only up to a scaled model size of factor 50. Here the intersection of the curves representing the adapter of Merschen and artshop is reached at around 22.000 blocks. As the artshop tool adapter imports much more data than the adapter presented by Merschen, it is expected that the artshop tool adapter experiences performance degradation to a certain degree when compared to an approach that only imports a subset of the information available within a model. Additionally, the tool

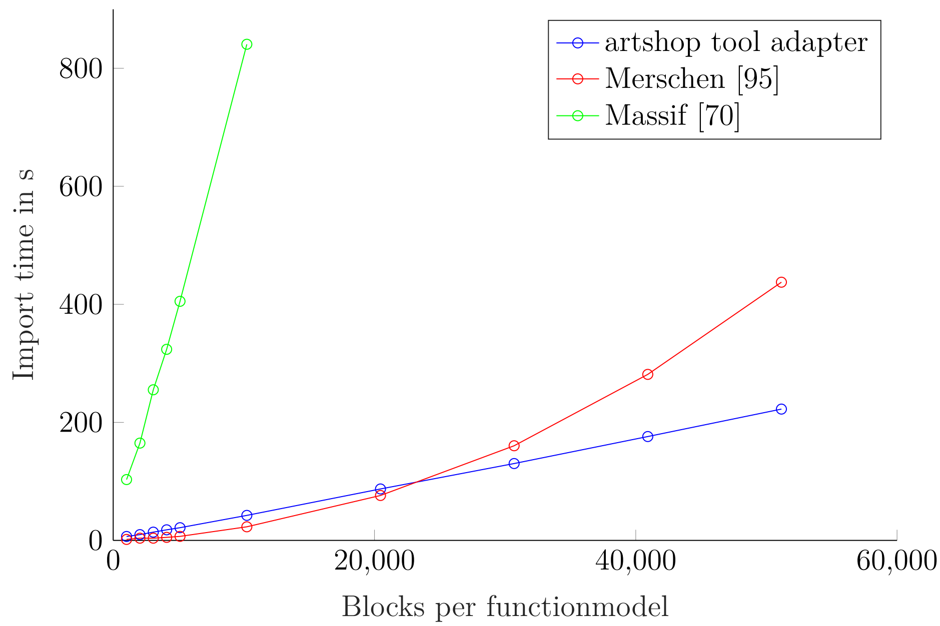


Figure 3.21: MATLAB/Simulink tool adapter import performance for functionmodels containing up to 50.000 blocks

adapter of Merschen is realized as a MATLAB script, which uses the underlying Java VM of MATLAB to execute Java code and store intermediate model data. In comparison, the artshop tool adapter reduces the amount of MATLAB code to be executed to a minimum and stores intermediate model data external to MATLAB, reducing the amount of load on the running instance of MATLAB/Simulink.

Export Functionality Both the artshop and Massif tool adapter also support the export of models imported via their respective import processes. Again, the MATLAB API is utilized by both tools to recreate imported models from the information available in the imported models. The features of the export functionality of both tools are listed in Table 3.3.

Both tool adapters support the export of blocks and line elements into a Simulink model. Neither adapter supports the export of Stateflow elements, with artshop only inserting an empty dummy stateflow block. Furthermore, the artshop tool adapter supports the recreation of library links as well as the export of annotation elements and the parameters of all supported elements including the Simulink model itself. Finally, the artshop tool adapter is able to preserve the SIDs of the blocks of a model.

We first tried to evaluate both approaches by using the imported models from the evaluation of the import procedure. Unfortunately, even for the initial version with around 1000 blocks, the Massif tool adapter failed with an exception, as it tried to write into a read-only attribute of a Subsystem block. Figure 3.22 shows how the artshop tool adapter scales during the export of the imported models.

Table 3.3: Comparison of supported export functionality of evaluated tool adapters

Functionality	artshop tool adapter	Massif [70]
Model parameter	x	-
Block elements/parameter/SID	x/x/x	x/partial/-
Line elements/parameter	x/x	x/-
Annotation elements/parameter	x/x	-/-
Stateflow elements/parameter	-/-	-/-
Library block insertion	x	n/a
Model reference insertion	-	n/a

In contrast to the import procedure, the export procedure of the artshop tool adapter takes much longer than the import procedure but still scales nearly linearly with regard to the size of the exported models. As the default parameters for a given model element are not known a priori, the exporter still has to export and set each individual parameter for each exported model element. This takes considerably more time than just reading (importing) all parameters, as done during the import procedure. Moreover, the artshop tool adapter detects exceptions thrown by MATLAB once a read-only parameter is written and skips all subsequent occurrences of these parameters for the given class of elements.

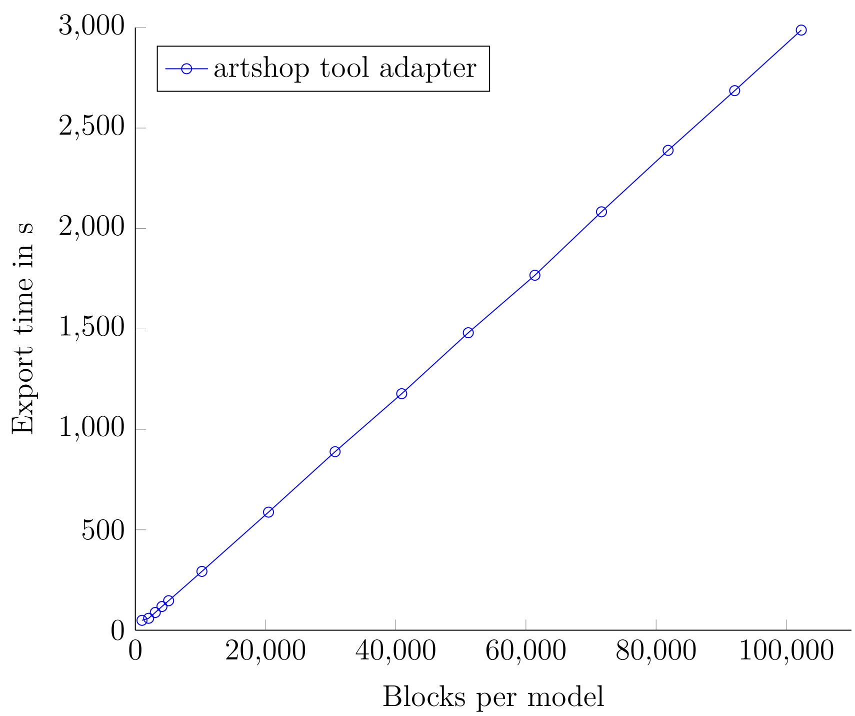


Figure 3.22: MATLAB/Simulink tool adapter export performance

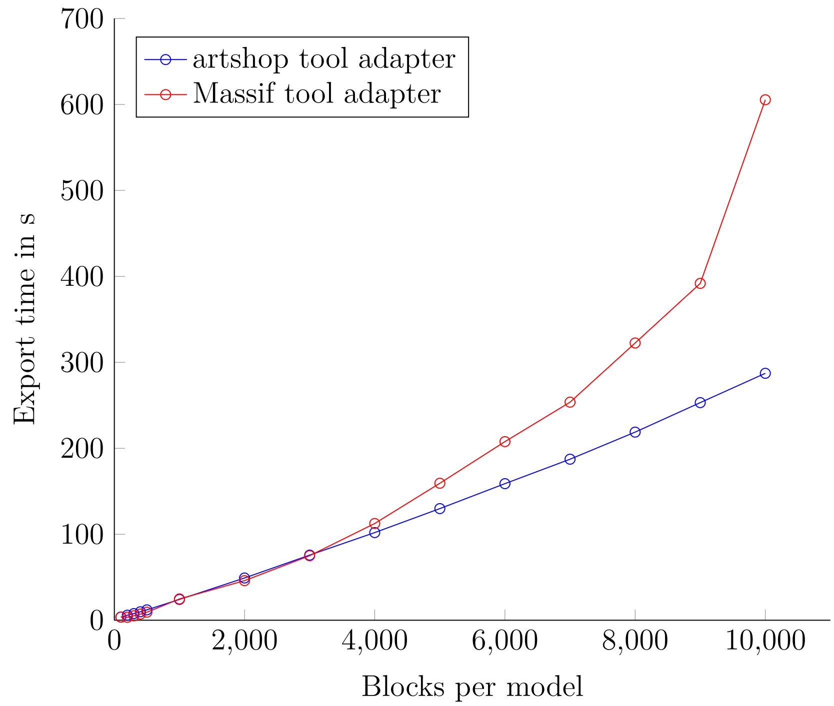


Figure 3.23: artshop tool adapter and Massif export performance

To evaluate the artshop tool adapter against the Massif framework, we tried to use the export function of Massif with multiple models from the MATLAB/Simulink example models and the model used during the evaluation presented by Merschen in [95]. Finally, we constructed a simple model fragment that consists of one Constant, three Gain and one Terminator blocks that are connected by lines. Massif was then able to export this model fragment to a new model. To evaluate the performance of the framework we duplicated this model fragment to create evaluation models with 100, 200, ..., 500, 1000, 2000, ..., 10000 blocks respectively. Each of these models are then imported by both the artshop and the Massif tool adapter and we measure the time it takes both adapters to export the imported models.

Figure 3.23 shows how both adapters scale during the export of the constructed example. Up to 3000 blocks, both adapters perform equally well with the export time scaling linearly with respect to the exported model size. After that, the performance of the Massif adapter diverges from the artshop adapter, which continues to scale linearly as it was the case for the initial batch of exports of the MAV model.

Besides the performance of both tool adapters, we also evaluated how much the exported model differs from the initial model by using the comparison utilities of the MATLAB/Simulink Model Comparison API. For the artshop tool adapter, only slight differences between the parameters of the actual and exported model could be detected. Differences were detected for parameters that cannot easily be manipulated, as they reside within specific MATLAB objects. The only difference found between the elements of both models are the position of labels on lines. The parameter controlling this property is not accessible via the MATLAB API and can only be accessed by parsing the accompanying model file. Models exported with the Massif framework differ by quite a margin from the initial model. Besides some differences in the model parameters, neither block nor line positions match the position of the initial model. This leaves the model in an unreadable state, with blocks and lines overlapping and being randomly distributed all over the model. Additionally, the SIDs of the model elements do not match anymore, as the SIDs depend on the order of insertion into the model that is not considered by the Massif framework.

While the export function of the artshop tool adapter is much more reliable and produces higher quality models than the Massif framework, further work has to be invested to refine the export of the remaining differing model parameters, model reference blocks and elements from MATLAB/Stateflow. Due to the errors occurring during the use of the Massif framework, we could not verify if Massif can insert library blocks or export model reference blocks.

MATLAB/Simulink Evaluation Models

Besides the import/export of the relatively simple MAV model, we have successfully used the tool adapter to import/export a multitude of academic and industrial models that include the use of Stateflow models, various library blocks, model reference blocks, masked subsystems and custom signal types. The models were retrieved from academic and industrial case studies, the Matlab File Exchange (a repository for MATLAB related files hosted by The Mathworks), MATLAB/Simulink demo models from versions ranging from MATLAB/Simulink 2008a to 2016b and openly available model repositories, e.g. the ReMoDD (Repository for Model-Driven Development) repository [55]. The size of these models ranges from a few hundred blocks in the case of simple example models to proprietary models created by an OEM from the automotive domain containing more than 100.000 blocks. In total, the adapter has been used to import over 2.200 models from the aforementioned locations.

A subset of these models will be referenced throughout this thesis during the evaluation of static analysis techniques on MATLAB/Simulink models. A few key indicators of these models are shown in Table 3.4. For the sake of completeness, the MAV model is also listed in this table. The DAS model is a control system for a driver assistance system, while the EL model contains the controller for an exterior light system of an automobile Both the DAS and EL model originate from an industrial case study from the automotive domain [111, p. 11-25]. The MAV model is a control model for an autonomous micro air vehicle described in [98]. The PI model has been taken from Matlab File Exchange and models the interpretation of a pedal in an automobile including a test harness and is described in [165]. Taken from an ongoing research project in a complex medical intensive care system, the ECLA model realizes a control model for an extracorporeal lung assist system (ECLA) applied to treat severe cases of the acute respiratory distress syndrome [17]. While the ECLA model does not originate from the automotive domain, the used

Table 3.4: Overview of MATLAB/Simulink models used throughout this thesis

Model name	Number of blocks	Number of lines	Hierarchy depth	Number of subsystems	Percentage of virtual blocks	Closed loop
DAS	1043	975 (47)	13	195	56.56	Yes
EL	1635	1791 (49)	10	197	58.10	No
MAV	1017	1069 (70)	5	96	50.24	No
PI	8224	9169 (272)	8	1038	42.92	No
ECLA	8746	9953 (516)	11	723	44.90	Yes

model paradigms are the same. Furthermore, the overall complexity of the model proved to be a valuable evaluation target, as we will show in later chapters.

Only the DAS and ECLA model are closed loop models. These kind of models use information calculated in a previous time step, introducing cyclic data dependencies, by feeding its output signals back to its inputs.

3.5.2 Evaluation of the IBM Rational DOORS Tool Adapter

Throughout this section, we show the evaluation results of the IBM Rational DOORS tool adapter presented in Section 3.3.2. Unlike the MATLAB/Simulink tool adapter, no proprietary tool adapter for IBM Rational DOORS was available for evaluation purposes but as we have implemented two different approaches to realize the import/export of formal modules from/to the tool, we will evaluate these approaches against each other. During the evaluation, we considers both the import and the export procedure and use artificially created data to evaluate both the OLE and Batch DXL dispatcher. We used IBM Rational DOORS 9.3 to perform the evaluation.

Evaluation of the import functionality To evaluate the import functionality, we created 13 formal modules containing 1, 100, 500, 1000, 2000, ... , 10000 objects respectively, with each of these objects having a fixed size of 1 byte by setting the default attribute *Object Text* of the respective DoorsObject to a string with length 1. This process was repeated three times with varying object sizes of 100 byte, 1 kilobyte and 10 kilobyte. All 52 formal modules were then imported using both DXL dispatching approaches.

Figure 3.24 shows the result of these import evaluation runs. The red graphs display results recorded using the OLE dispatcher, while the blue graphs display the results recorded with the batch dispatcher. It can be noted that the import performance for both dispatchers is nearly identical for the runs targeting the modules containing objects of size 1 byte and 100 byte with the OLE dispatcher having a slight edge in comparison to the batch dispatcher. These modules can be imported in 1-5 seconds. Starting with the import of modules containing objects of size 1 kbyte, the performance of both dispatchers starts to degrade with the OLE dispatcher again getting an edge over the batch dispatcher with increasing module size. Neither approach takes more than 10 seconds. The performance

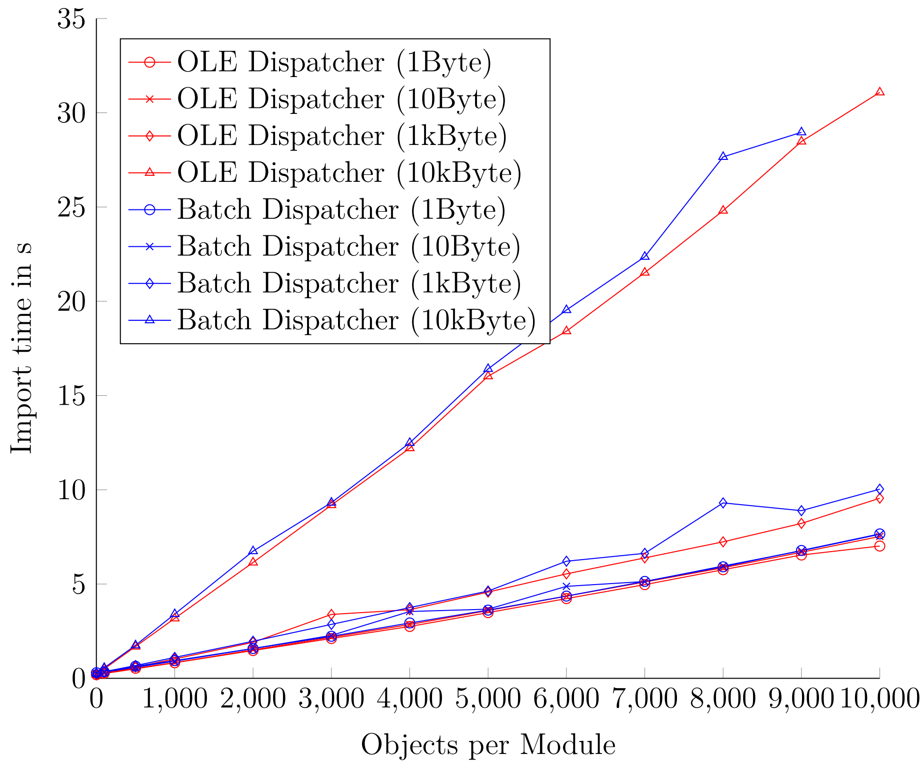


Figure 3.24: IBM Rational DOORS tool adapter import performance

impact becomes even more notable when the size of the individual objects contained in a module are increased to 10 kilobyte. Again both dispatchers start with similar performance characteristics, with the batch dispatcher performing slightly worse than the OLE dispatcher, taking at most 31 seconds. It has to be noted that the last data point of the batch dispatcher is missing as we were not able to finish the import of the formal module containing 10000 objects with the Batch dispatcher due to an internal exception within DOORS itself. In total, the OLE dispatcher performs slightly better during the import procedure than the Batch dispatcher while at the same time having the upside of using the credentials of a user that has started an instance of the DOORS client application.

Evaluation of the export functionality The artificial data used in the previous paragraph was created using the export functionality of the tool adapter. Again, the OLE and batch DXL dispatcher can be used to issue commands to the DOORS application. A module is exported by first creating a new module as an instance of the model representation described in Section 3.3.2, adding its containing objects and setting all attributes to the desired properties. A DXL command that creates these instances within the DOORS client application is derived from the created model instances and handed to the respective DXL dispatcher. Figure 3.25 shows the results of the export procedures

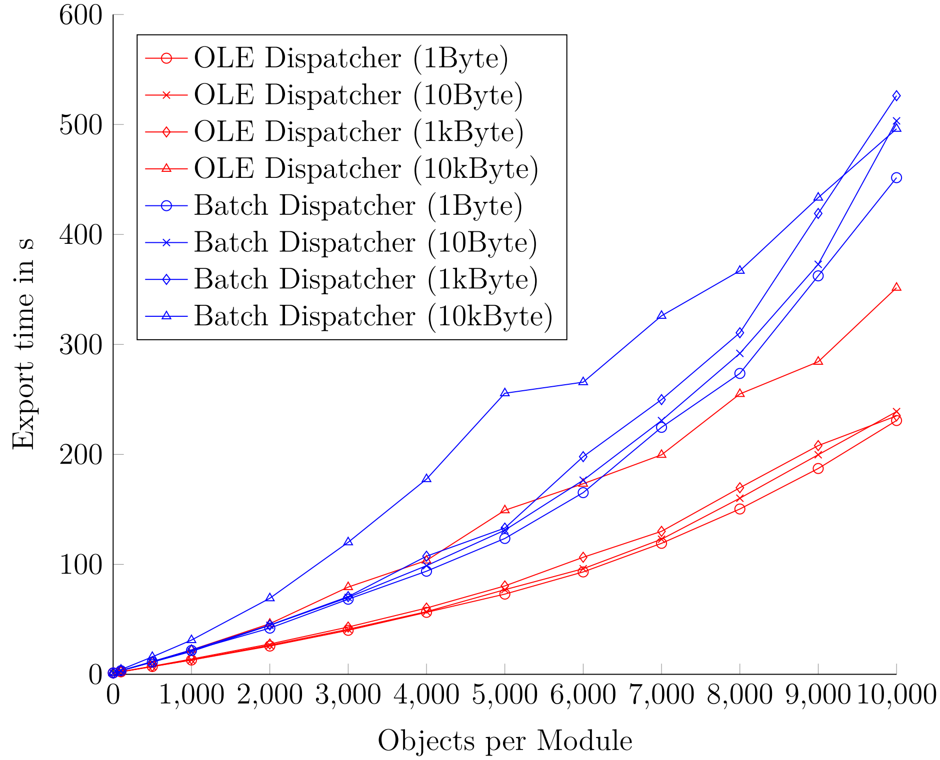


Figure 3.25: IBM Rational DOORS tool adapter export performance

for these modules for both the OLE (red) and Batch (blue) dispatcher. While the import procedure of both dispatchers took at most 31 seconds to import the biggest module, the export procedure takes up to 351 seconds with the OLE dispatcher and 496 seconds with the Batch dispatcher. This discrepancy between the import and export procedure originates from the fact that the DXL scripts used throughout the import procedure simply query data stored in the DOORS server, while new objects have to be created by the scripts of the export procedure. Therefore, the complete content of each individual object has to be sent to the server that stores it in its underlying database. Unlike the performance graphs of the import procedure, all graphs shown in Figure 3.25 display an exponential function, which is related to the memory management of the DXL scripting environment in the DOORS client. Overall, the OLE dispatcher performs much better than the Batch dispatcher and only suffers performance degradation for the modules containing objects with a size of 10 kByte. However, the Batch dispatcher performs and scales much worse than the OLE dispatcher, which is related to the construct used to perform inter-process communication with the DXL server created by the DOORS client started in batch mode.

After the evaluation of both DXL dispatchers, it can be concluded that the OLE dispatcher should be used as the default DXL dispatcher in the IBM Rational DOORS tool adapter. Performance-wise the OLE dispatcher outperforms the Batch dispatcher

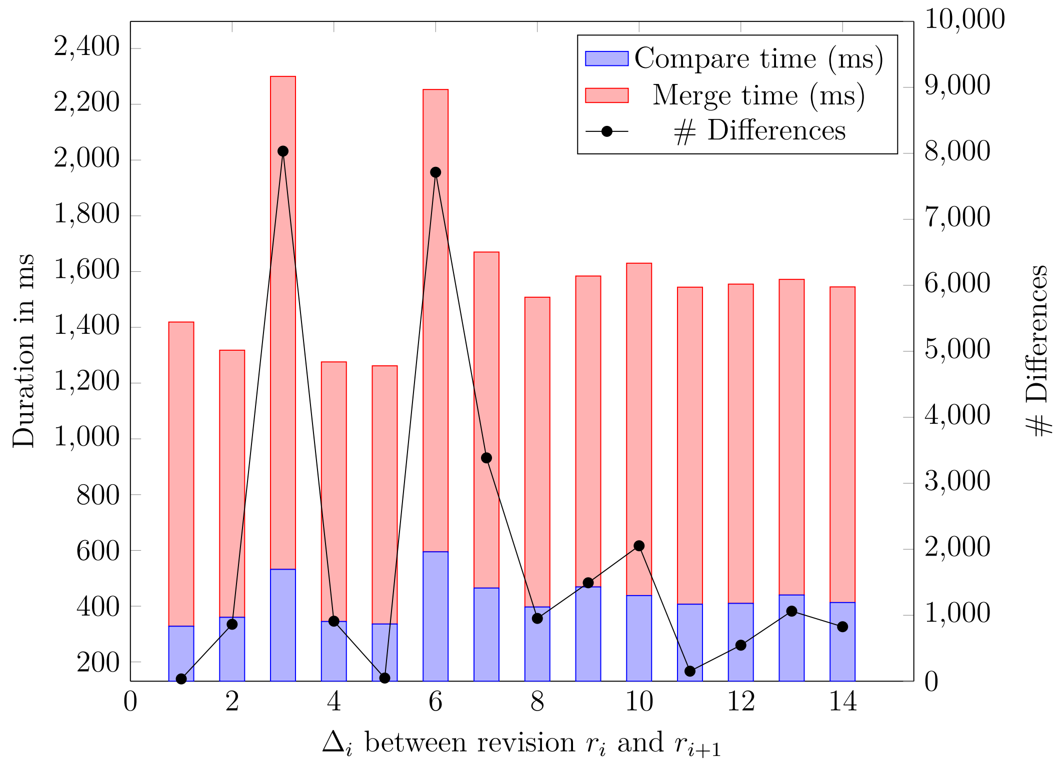


Figure 3.26: Performance of the synchronization mechanism across all evaluated model Δ and detected differences

during import and export of module data and operates slightly more stable than the batch dispatcher. As a result, the OLE dispatcher was chosen as the default dispatcher in the implementation of the Doors tool adapter.

3.5.3 Evaluation of the Model Synchronization Mechanism

To evaluate the model synchronization mechanism we used a proprietary set of control models from MATLAB/Simulink taken from an industrial automotive-related project realized at the chair for embedded software. The control model is responsible for controlling the movement of a prototypical mobile device based on environmental inputs. It consists of 4 distinct MATLAB/Simulink models that include each other via model reference blocks. In total, 15 revisions of the control model were available for the evaluation. The models evolved over time starting at around 250 blocks in the first revision with the final revision contained about 400 blocks.

We evaluated the synchronization mechanism by importing the first revision of the model and then subsequently synchronizing it with all following revision. For each synchronization step we recorded the time it took to analyze the differences between the current revision r_i and the new one r_{i+1} and the time it took to merge the found differences into revision r_i. Moreover, we stored the amount of differences detected by the mechanism to relate them to the time consumed by the individual synchronization steps. After all differences have been merged and r_i has been updated to r_{i+1}, we imported the model of r_{i+1} again, perform another comparison between the updated model and the newly imported model and execute a deep-equality check on these models. This checks if each object in the synchronized model is equal to its corresponding object in the newly imported model. Finally, we save both models on the file system and perform a text comparison of the files created during the serialization of both models. None of these checks were able to detect a difference between a synchronized and newly imported version of the control model. Therefore, we conclude that the synchronization mechanism correctly preserves the structure of a model while synchronizing it with an evolved version of the same model.

Figure 3.26 shows the performance of the synchronization mechanism across the 14 Δ between the 15 model revisions of the evaluated model. The durations of the individual phases of the synchronization mechanism are shown on the left-hand y-axis while the amount of differences are displayed on the right-hand y-axis. It can be noted that the time it takes to calculate and merge the differences between two model revisions directly depends on the amount of differences between two model revisions as it can be seen for Δ_3 and Δ_6. Further, it can be noted that regardless of the amount of detected differences, the time to merge the differences between two model revisions at least takes one second. This is related to the custom handling of the attribute introduced by the artshop metamodel, which requires a pre-processing routine that analyzes the complete model to capture attribute mappings to their corresponding AttributableElement instances as these are excluded from the comparison performed by EMFCompare. When configuring EMFCompare to also analyze the attributes of a model, comparison and merge times of EMFCompare increase to over 30 seconds in the worst case, while at the same time not producing equal models due to the shortcomings of the merge of multi-valued attributes mentioned in Section 3.4.2. Thus, the overhead of the custom solution is acceptable with respect to the performance degradation of the unaltered EMFCompare framework. Overall, even for a huge amount of differences, the synchronization procedure takes around 2.4 seconds not including the import of the updated model revision.

3.6 Conclusion and Future Work

In this chapter, we have presented the base components of the artshop framework, which is a research platform for the analysis of model-based software artifacts. We defined a metamodel able to represent common properties of model-based software artifacts as well as specializations of this metamodel for three specific development artifacts: functionmodels from MATLAB/Simulink, formal modules from IBM Rational DOORS and feature models, family models and variant description models from pure::systems pure::variants. A tool adapter was created for each model refinement of the metamodel that allows the import and instantiation of these models based on real-world artifacts. These artifact instances can be stored in a model repository offering version control and querying capabilities. The metamodel further includes concepts to express traceability links between imported model elements as well as links to artifacts outside of the repository. Furthermore, a synchronization mechanism has been integrated to synchronize models loaded in the repository with respect to changes made to the initially imported artifacts. This allows the incremental application of changes, while at the same time preserving existing traceability links in the repository.

The evaluation of the presented components indicates that the presented approaches can be used to import and store models from academic and industrial contexts and scale well enough to support the complexity of real-world industrial models. Therefore, the base components of the artshop framework, artconnect, the repository and artshop.core provide a solid foundations for the analysis of model-based software artifacts, as imported models have a well-defined and documented structure that allows the access of all imported properties for use in various syntactic and semantic intra- and inter-artifact analysis techniques. Besides the techniques described in subsequent chapters, the framework was adopted by Dernehl et al. to implement a static value range analysis on MATLAB/Simulink models [40, 41, 42, 43, 44] as well as in a multitude of bachelor and master's theses [15, 66, 87, 102, 133, 136, 159] that were either directly or co-supervised by myself. In addition, a traceability link discovery tool that is currently being developed at our chair also uses the foundations of the artshop framework.

3.6.1 Future Work

In future work, the base framework could be extended by integrating an OSLC interface into the tool. This could potentially increase the interoperability of the framework with other tools, as currently no external API is exposed by the framework itself. The development of an external API or a batch mode of the framework to fetch data or trigger implemented analyses from external applications could further broaden the use-cases of the framework.

Currently, the tool adapter for MATLAB/Simulink requires an activated copy of MATLAB/Simulink to import a model as it significantly eases accessing static and compile-time model data as well as library and model reference block resolution. A hybrid approach such as the MeMo tool suite [115] developed by Reicherdt imports models directly from their model files by parsing their contents and enriching these imported models by compile-time attributes from Simulink. Such an approach could also be adopted for the artshop tool adapter and further refined by implementing the automatic inference of compile-time attributes from the model files, e.g. type inference, sorted order and execution contexts. A simple type inference algorithm for MATLAB/Simulink models has already been proposed by Preoteasa et al. [112]. It needs to be evaluated if the increased effort needed to manually implement and maintain the model file parsers, model/library resolution techniques and the inference of compile-time attributes for all available and future versions of MATLAB/Simulink is worth the performance advantages over the approach presented in this chapter.

A limitation of the tool adapter for MATLAB/Simulink is that it does not support the import of lines created between blocks of the SimPower toolbox. This toolbox allows the modeling of electrical circuits in MATLAB/Simulink. Therefore, lines connecting blocks from this toolbox carry electrical current and are undirected. Correctly representing the lines from the SimPower toolbox should be addressed in future work.

Adding new tool adapters should also be considered as part of future work. With architectural models slowly gaining traction within the automotive domain, modeling languages such as the UML and SysML can also be used to derive corresponding MATLAB/Simulink models directly from the components described in these tools, as it is currently possible by various implementations of the AUTOSAR standard. The adoption of other data flow languages similar to MATLAB/Simulink, e.g. SCADE or ASCET, might further increase the viability of the framework to other domains, e.g. the avionic domain.

The views provided by the framework might also be improved in future work. In particular, views related to the evolution of models are currently only prototypical implementations showing the history of elements including their changes as it is shown in Figure 3.19. Further integration of this information within the other views provided by the tool adapters as well as graphical change views could enhance the usability of history information captured by the framework.

While we tested the synchronization mechanism on a case study containing 15 model revisions, we could not assess its performance on a large-scale industrial model, as such a model was not available for evaluation purposes. In the future, the model evolution techniques should be evaluated in a longer and more complex case study.

4 Consistency Checking in Software Product Lines

A typical problem in product and software development is the consistent handling of traceability information between problem and solution space in the context of software product lines [10]. To guarantee the correct derivation of a product from a product line, variability needs to be documented consistently across all involved artifacts. Due to the amount of involved artifacts and their increasing complexity, manual consistency checking of these variability documentations is becoming more costly and economically infeasible. At the same time, the consistency of the variability documentation is an important quality criterion, as errors induced by inconsistencies might be propagated to additional artifact elements [162]. The manual detection and correction of such errors is an error-prone and time-intensive process. Therefore, automatic analyses for the detection of inconsistencies are required that can be integrated into existing development processes. Suchlike analyses need to support the recognition of variability information in the artifacts of the product line as well as the detection of inconsistent mappings of this information.

4.1 Approach

Throughout this chapter, we introduce a method for the automatic extraction of variability information and inter-artifact consistency checks for product lines managed by the tool pure::variants (see Section 3.3.3). As pure::variants documents variability in the form of feature and family models, the presented methodology targets features as introduced in Section 2.3.2. Features are extracted from the variability documentation of the artifacts of the product line, i.e. family models, and validated against each other by utilizing additional inter- and intra-artifact relationships, e.g. test cases associated to requirements. The method includes the detection, categorization and resolution of inconsistent mappings of variability information attached to the artifacts of the product line.

4.1.1 Related Work

To tackle the complexity involved in performing consistency checking on the dependencies and relationships in a real-world industrial product line, Vierhauser et al. propose an incremental consistency checking approach to increase the responsiveness of the consistency checker [156, 157]. The approach can detect inconsistencies of the variability documentation and its implications in and between all levels of a software product line. For each inconsistency type, the authors define constraints that can be checked by an incremental consistency checker inspired by the work of Egyed [48]. This enables the change based checking of large-scale variability models across all levels of the product line.

Another approach for incremental consistency checking for UML models with delta-based variability modeling is presented by Kowal et al. [86]. In contrast to the approach of Vierhauser et al. [156, 157], the authors do not define inconsistencies based on the levels of the product line but rather on different perspectives on the UML-Model, i.e. *workflow, architectural* and *behavior*. Consistency rules are categorized as intra-perspective affecting a single perspective, inter-perspective rules that affect all perspectives of a derived variant and cross-variant rules affecting the complete product line. Rules have been defined manually and have been evaluated in a case study involving an automation system called the *Pick and Place Unit* described by Legat et al. [90].

An approach for feature-based consistency checking has been presented by Cmyrev et al. [29]. The authors present an approach to check if the feature mappings of requirements and test cases are consistent to each other. Detected inconsistencies are classified into three categories: contradictoriness, redundancy and incompleteness. The feature mapping of a requirement is contradictory regarding its associated test cases if the requirement is mapped to features disjoint to the feature mapped to its test cases. The paper formalizes the approach of detecting these inconsistencies. A prerequisite for this approach is that the associations between requirements and test cases are given and complete. An extension to this approach is given by Wiechowski et al. in [162]. The authors extend the scope of the analysis by function models from MATLAB/Simulink and introduce the concept of cliques, which bundle semantically associated artifact elements with each other. Inconsistencies within the feature mappings are detected between the elements of a clique. In addition, the paper proposes a set of resolution operations for the inconsistency categories proposed in [29]. These operations can be used for the semi-automatic resolution of inconsistencies found by the analysis.

4.1.2 Bibliographic Notes

The approach presented in this chapter extends the methodology presented in the PhD. thesis of Merschen in [95] that was also published in [162]. It was developed during the project *SPES_XT* that has been funded by the German Federal Ministry of Education and Research (BMBF) [56]. The development of the methods presented in this chapter were driven by the artifacts contained in the case studies developed by Daimler [111, p. 11-25] and therefore focus on a specific set of tools.

4.2 Preliminaries

We will first introduce basic concepts and terminology related to software product lines not yet covered by the foundations presented in Section 2.3.

Throughout this chapter, we will refer to the artifacts of a software product line by the following definition.

Definition 4.1 (Artifacts of a Product Line).
The artifacts $A_1, .., A_n$ that are used in a product line PL are denoted by the set $\Lambda_{PL} = A_1, .., A_n$. Each artifact A_i itself represents a set of artifact elements it contains.

The construction of a feature mapping for arbitrary elements of the artifacts of a product line is an important part of the methodology we have presented in [162]. A feature mapping describes how features of a feature model are mapped to the artifact elements of the product line.

Definition 4.2 (Feature Mappings for Artifacts).
Let F be the set of features contained within a feature model. A feature mapping $f : A \rightarrow \mathcal{P}(F), A \in \Lambda_{PL}$ maps a set of artifact elements $A' \subseteq A$ to a set of features $F' \subseteq F$.

$$f(A') = F' = \{f_1, .., f_n\}$$

With rising size and complexity of the underlying product line, the manual creation and annotation of features to their respective artifacts, as suggested in [95], is a time-intensive and error-prone process. Therefore, a pre-processing procedure that automatically derives the current feature mappings from the available artifacts of the product line can increase the quality of the feature mapping used for consistency checks of the variability documentation. Such a procedure needs to analyze the structure and links of the variability documentation of the problem space to the artifacts in the solution space.

4.3 Automatic Feature Derivation

Mappings from the variability documentation to the artifacts of the solution space, e.g. mapping a feature to a requirement, can be documented in diverse ways, depending on the tools and methods used to manage the product line. In this thesis, we will focus on family models created and managed by the tool pure::variants to document feature mappings to development artifacts. These models define the mapping of an artifact element to a feature defined in a feature model, which allows the derivation of a specific product variant in combination with the variant description model (see Section 3.3.3). A family model $F(A_{Fam})$ is associated to one artifact A_{Fam} and contains a tree of components representing a subset of the artifact elements from A_{Fam}.

Definition 4.3 (Family Model).
A family model $F(A_{Fam}) = (C, E, r_c, \alpha, f_{Fam})$ for an artifact A_{Fam} has the properties:

- *C: Finite set of components*
- *E: Finite set of edges so that holds $E \subseteq \{(u, v)|u, v \in C\}$*
- *α: Artifact mapping function $\alpha : C \rightarrow A_{Fam}$*
- *r_c: Root component*
- *f_{Fam}: Mapping function $f_{Fam} : C \rightarrow \mathcal{P}(F)$*

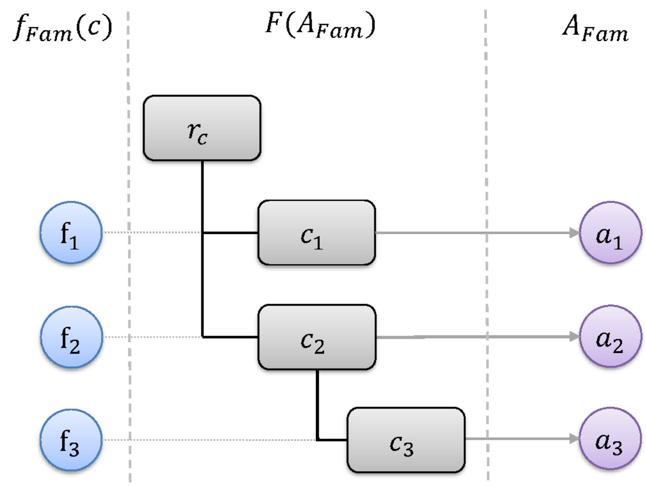


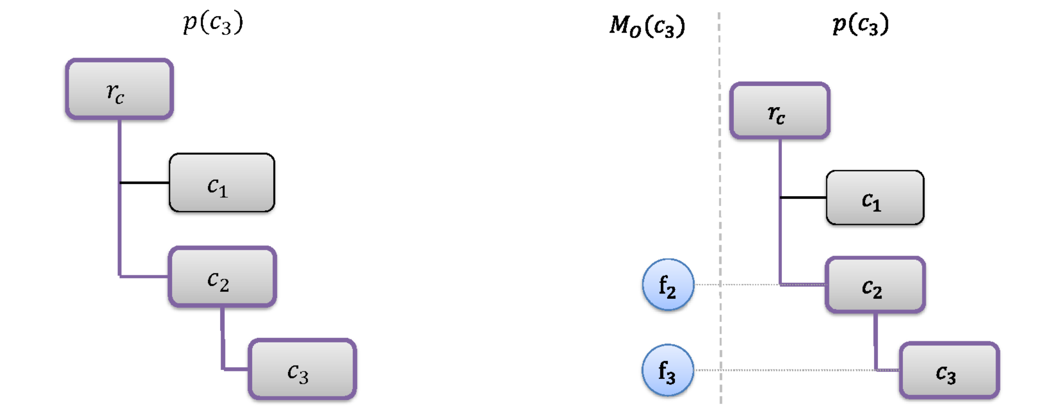
Figure 4.1: Exemplary illustration of a family model for an artifact A_{Fam} (figure taken from [56])

Figure 4.1 shows the structure of an exemplary family model for an artifact A_{Fam} with four components. Each component of a family model can be connected to an artifact element of the underlying artifact A_{Fam} using the mapping function α. Furthermore, a component is associated with a set of features specified by the mapping function f_{Fam}, which connects a component to a set of features from the associated feature model. A component and its sub-components are only considered during product derivation if their associated feature restrictions are satisfied by the feature configuration used for product derivation defined in the variant description model. Thus, a family model might impose additional constraints and dependencies in addition to the ones defined in the underlying feature model. Both mapping functions are shown by the lines connecting the components with the elements from A_{Fam} in the case of α and from the components to the features of the feature model for the feature mapping function f_{Fam}.

Based on the mapping function α, it is possible to collect the obligatory features of an artifact element $a \in A_{Fam}$ that need to be selected in a variant description model V to include a component as part of the product defined by V. The obligatory features are computed by collecting all features attached to the components that are encountered on the path $p(c)$ from the root component r_c to c with $\alpha(a) = c$. We will denote the path $p(c)$ as the component path of c.

Definition 4.4 (Component Path).
A component path $p(c)$ in a family model $F(A) = (C, E, r_c, \alpha, f_{Fam})$ is defined as a sequence of $c_1, .., c_n$ so that holds:

- $1 \leq i \leq n : (c_i, c_{i+1}) \in E$
- $c_1 = r_c$
- $c_n = c$



(a) Component path to component c_3 (b) Obligatory features of component c_3

Figure 4.2: Component path and obligatory features of component c_3 (figures taken from [56])

Figure 4.2a shows the component path from the root component r_c, over component c_2 to component c_3 from the family model shown in Figure 4.1. All other components are ignored.

Based on the definition of the component path in Definition 4.4, the obligatory features of a component c, and therefore also for an artifact element a with $\alpha(c) = a$, can be determined by collecting all features attached to the components on the component path $p(c)$.

Definition 4.5 (Obligatory Features).
The obligatory features of a component $M_O(c)$ are defined as

$$M_O(c) = \bigcup_{i=1}^{n} f_{Fam}(c_i) \textit{ with } c_i \in p(c)$$

On the left-hand side of Figure 4.2b, the obligatory features $M_O(c_3)$ of component c_3 are displayed. The information obtained from the obligatory features of a component can be used to construct a feature mapping for the artifact elements associated to the components of the family model via the mapping function α. An artifact element a is then mapped to the obligatory features $M_O(c)$ of its associated component c with $\alpha(c) = a$, if such a component exists. The mapping can then be defined as follows:

Definition 4.6 (Feature Mapping from Family Model).
The feature mapping $f_{(F,A)}(a)$ from a family model $F(A)$ and an artifact A of an artifact element $a \in A$ is defined as

$$f_{(F,A)}(a) = \begin{cases} M_O(c), & \textit{if holds: } \exists\, c, c \in C \wedge a(c) = a \\ \emptyset, & \textit{otherwise} \end{cases}$$

Following Definition 4.6, the mapping is dependent on the information of the family model and maps all features required for the derivation of a component c to its corresponding artifact element. The mapping could also be altered to map only the features directly associated with c. We include this as an option during the actual derivation procedure. As a family model only contains the variability information of one artifact $A \in \Lambda_{PL}$, multiple family models need to be taken into consideration when deriving the feature mappings of all artifacts in Λ_{PL}. Before using the feature mappings during the methodology described in [162], the consistency of the mappings obtained from all family models can be checked with the help of additional traceability information connecting the artifacts of the product line, similar to the approach presented by Cmyrev et al. [29]. A major difference to the approach of Cmyrev et al. is that the structure of the family models used to create the feature mappings contain additional information in the form of the obligatory features of an artifact element, i.e. the features needed to include an artifact element during product derivation. If the obligatory features of two artifact elements a and b, with $a \in A, b \in B, A, B \in \Lambda_{PL}$, connected by additional traceability information differ from each other, a might be included in a product while b which a might depend on, is not included, leading to an inconsistent state in the derived product. Thus, our method includes the dependencies and constraints imposed by the family model into the consistency checking process in addition to the constraints and dependencies considered by [29, 162].

4.4 Consistency of Connected Feature Mappings

As mentioned in the previous section, a consistency check of the derived feature mapping from Definition 4.6 can be applied if multiple family models exist for the artifacts Λ_{PL} and there exist inter-artifact traceability links connecting semantically associated elements with each other.

Definition 4.7 (Inter-artifact Relationships)**.**
Given two artifacts A, B then the inter-artifact relationship between the elements of these artifacts is defined as

$$g_{A,B}(a, b) = \begin{cases} 1, & \textit{if } (a, b) \in E_{A,B} \\ 0, & \textit{otherwise} \end{cases}$$

$$\textit{with } E_{A,B} \subseteq \{(a, b) \mid a \in A \wedge b \in B\}$$

The actual relationship expressed by $g_{A,B}$ is stored as a set of 2-tuples $E_{A,B}$. The contained 2-tuples (a, b) represent a connection between the artifact elements $a \in A$ and $b \in B$. In combination with Definition 4.5, the defined inter-artifact relationship can be used to determine if the feature mapping of two family models are consistent to each other. To check the consistency of the feature mappings derived from a component c from a family model $F(A)$ against the feature mappings of an associated family model $F'(B)$, the set of related components from $F'(B)$ needs to be computed by using the inter-artifact relationship between the artifacts A and B from Definition 4.7.

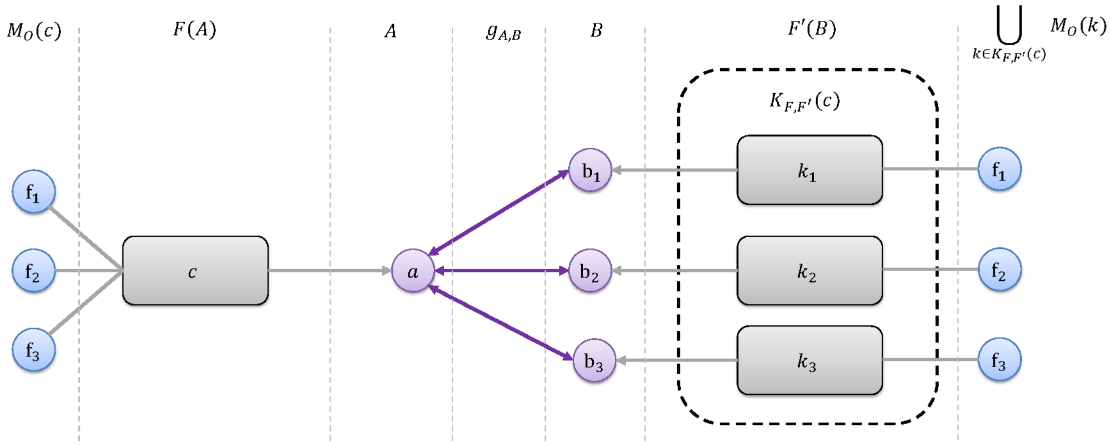


Figure 4.3: Sketch of a feature consistent component (figure taken from [56])

Definition 4.8 (Related Components)**.**
Let $F(A) = (C, E, r_c, \alpha, f_{Fam})$ and $F'(B) = (C', E', r'_c, \alpha', f'_{Fam})$ be two family models. The set of related components K of a component $c \in C$ with $K \subseteq C'$ is defined as

$$K_{F,F'}(c) = \{k | \exists a, \exists b \ (g_{(a,b)} \wedge (\alpha(a) = c) \wedge (\alpha'(k) = b))\}$$

Following this definition, a component c from $F(A)$ is related to another component k from $F'(B)$ if the artifact elements associated to c and k are connected by an inter-artifact relationship.

Based on this component relationship we can now define an invariant describing a feature consistent component.

Definition 4.9 (Feature Consistent Component)**.**
Given a family model $F(A) = (C, E, r_c, \alpha, f_{Fam})$ then a component $c \in C$ is feature consistent to its related components $K = K_{F,F'}(c)$ if it holds

$$K \neq \emptyset \wedge M_O(c) = \bigcup_{k \in K} M_O(k)$$

Figure 4.3 shows a component c and its related components that satisfy the invariant from Definition 4.9. All features present in the set of obligatory features $M_O(c)$ of component c are also present in the union of obligatory features from the set of related components $F_{F,F'}(c)$.

The defined invariant for feature consistent components is only checked for components c from $F(A)$ that have related components in $F'(B)$. It is recommended that the consistency check be performed with one of the family models as a reference model, which is assumed to be correct. Depending on the family model that is chosen as the reference model, different types of inconsistencies can occur which are discussed in the following section.

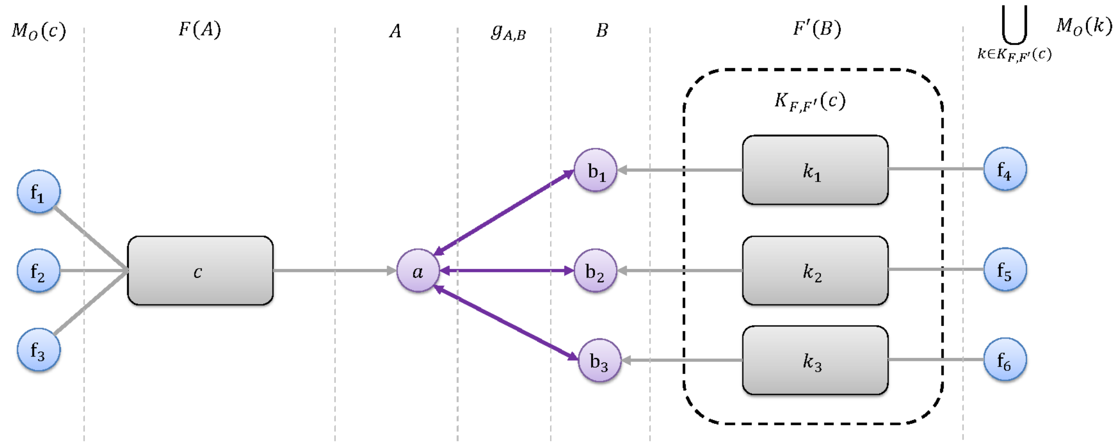


Figure 4.4: Contradictory feature mapping between c and $K_{F,F'}(c)$
(figure taken from [56])

4.4.1 Categorization of Inconsistencies Between Related Family Models

If the invariant of Definition 4.9 is not satisfied for a given component, an inconsistency is most likely to be present in the analyzed family models and thus in the variability documentation of the product line. Inconsistencies can be tracked back to the feature restrictions of the components in the family model and are detected by comparing the obligatory features of two related components.

As in our previous work [162], we differentiate between three types of inconsistencies of the compared feature sets obtained from the related components of two family models: *contradictory*, *incomplete* and *redundant*. In the following sections, we will describe these three inconsistency types and assume that the family model F is chosen as the reference family model for feature comparison, i.e. inconsistencies are detected against the feature mappings present in F. In addition, we assume the inter-artifact relationship from Definition 4.7 to be correct.

Inconsistency by Contradictoriness

The obligatory features of a component c and its related components $K_{F,F'}(c)$ contradict each other if they are disjoint. An example for an inconsistency by contradictoriness between the obligatory features of a component c and its related components $K_{F,F'}(c)$ is shown in Figure 4.4, as none of the features mapped to component c is contained in the obligatory features of $K_{F,F'}(c)$.

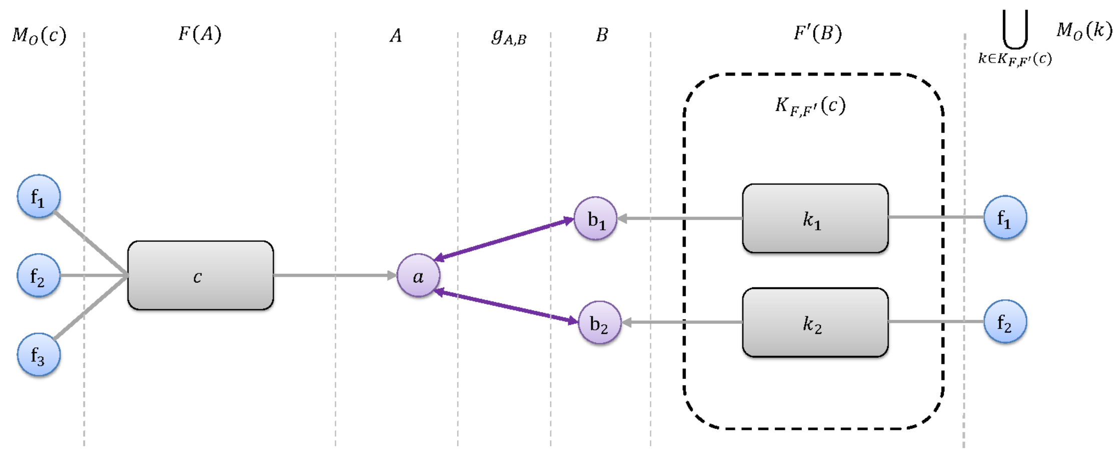


Figure 4.5: Incomplete feature mapping between c and $K_{F,F'}(c)$ (figure taken from [56])

Definition 4.10 (Inconsistency by Contradictoriness)**.**
An inconsistency by contradictoriness between the obligatory features of a component c and its related components $K = K_{F,F'}(c)$ is present if holds

$$\left| M_O(c) \cup \bigcup_{k \in K} M_O(k) \right| \geq 1 \wedge \left(M_O(c) \cap \bigcup_{k \in K} M_O(k) \right) = \emptyset$$

Inconsistency by Incompleteness

The obligatory features of a set of components $K_{F,F'}(c)$ related to component c are incomplete in regard to $M_O(c)$ if they represent a subset of $M_O(c)$. Figure 4.5 shows an example for an inconsistency by incompleteness, as the feature f_3 is missing among the obligatory features of $K_{F,F'}(c)$.

Definition 4.11 (Inconsistency by Incompleteness)**.**
An inconsistency by incompleteness between the obligatory features of a component c and its related components $K = K_{F,F'}(c)$ is present if holds

$$\left| M_O(c) \cup \bigcup_{k \in K} M_O(k) \right| \geq 1 \wedge M_O(c) \supset \bigcup_{k \in K} M_O(k)$$

Inconsistency by Redundancy

If the obligatory features in the set of related components $K_{F,F'}(c)$ for a component c are a superset of $M_O(c)$, we assume them to contain one or more redundant features. An example for such an inconsistency by redundancy is shown in Figure 4.6, where component $k_3 \in C'$ is mapped to feature f_4 that is not present in $M_O(c)$. If k_3 is related to another component $c' \in C$ that requires f_4, the redundancy regarding component c is not treated as an inconsistency by redundancy. This is reflected by the second part of the following definition.

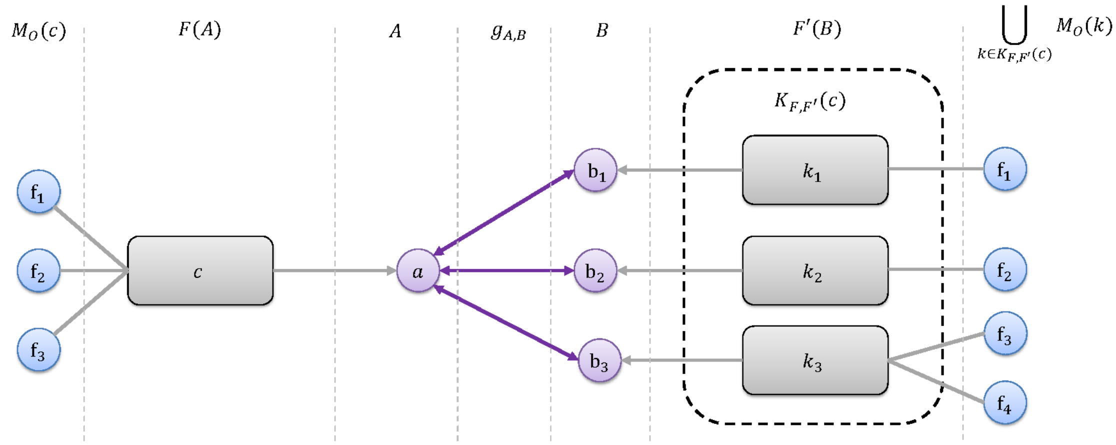


Figure 4.6: Redundant feature mapping between c and $K_{F,F'}(c)$ (figure taken from [56])

Definition 4.12 (Inconsistency by Redundancy).
An inconsistency by redundancy between the obligatory features of a component c and its related components $K = K_{F,F'}(c)$ is present if holds

$$\left| M_O(c) \cup \bigcup_{k \in K} M_O(k) \right| \geq 1 \wedge M_O(c) \subset \bigcup_{k \in K} M_O(k)$$

and there exists no set of components $\overline{C}$ from F so that holds

$$(K \cap \overline{K} = \emptyset) \wedge (((\cup_{k \in K} M_O(k)) \setminus (M_O(c))) \cap (\cup_{\overline{c} \in \overline{C}} M_O(\overline{c}))) \subseteq (\cup_{\overline{c} \in \overline{C}} M_O(\overline{c}))$$

$$\textit{with } \overline{K} = \bigcup_{\overline{c} \in \overline{C}} K_{F,F'}(\overline{c})$$

After we have described the invariants of the three types of inconsistencies, the next section will cover how detected inconsistencies can be resolved.

4.4.2 Inconsistency Resolution

The inconsistencies described in the previous section can have different causes. As the inconsistencies are always described relating to a family model $F(A)$, the cause of the inconsistencies lies in the feature mappings of the components of family model $F'(B)$. In reality, errors may also be present in the inter-artifact relationship described in Definition 4.7 and the feature mappings of $F(A)$. These types of errors are not handled within this work but can still be detected based on the inconsistencies described in the previous section.

Resolution of inconsistencies by contradictoriness To resolve the inconsistency imposed by disjoint obligatory features of component c and its related components $K_{F,F'}(c)$, the user needs to be involved because expert knowledge is required. One possibility to remove such an inconsistency is to either correct the feature mappings of all components residing on the component path of c or $K_{F,F'}(c)$ manually or to perform this process semi-automatically by choosing either F or F' as the reference model and reassign the corresponding feature mappings of the components in the other family model.

Resolution of inconsistencies by incompleteness Inconsistencies on the basis of missing features in the set of obligatory features of $K_{F,F'}(c)$ can be resolved by adding these missing features $(\cup_{c \in C} M_O(c)) \backslash (\cup_{c' \in C'} M_O(c))$ to the feature mappings of the component paths of $K_{F,F'}(c)$. This process also has to be performed semi-automatically by preparing the set of features that need to be assigned to the component paths of al l components in the set of components $K_{F,F'}(c)$.

Resolution of inconsistencies by redundancy Redundant features among the set of obligatory features of $K_{F,F'}(c)$ are the only kind of inconsistencies that can be resolved automatically. Removing the set of features $(\cup_{c \in C} M_O(c) \cap (\cup_{c' \in C'} M_O(c'))$ from the mappings of the component paths of $K_{F,F'}(c)$ removes the redundant features.

Choosing to apply the resolution operations described in this section might induce other inconsistencies within the analyzed family models as changing the feature mappings has implications on the validity of the derived mappings of other components. Therefore, resolution operations need to be used with care and resulting inconsistencies need to be resolved until a consistent state is reached. If a consistent state across all family models of the product line is reached, a generic feature mapping can be derived from the feature mappings of these family models.

4.4.3 Construction of Feature Mappings

To construct a generic feature mapping for the artifact elements of a product line, all family models need to be taken into account. Thus, we use Definition 4.6 to create a feature mapping for individual artifact elements as defined in Definition 4.2.

Definition 4.13 (Construction of a Generic Feature Mapping).
A feature mapping $f : A \to \mathcal{P}(F)$ $A \in \Lambda_{PL}$ based on a set of family models F_S is defined as

$$f_{(F_S,A)}(a) = \begin{cases} f_{F(A),A}(a), & \textit{if } \exists\ F(A), F(A) \in F_S \wedge A \in \Lambda_{PL} \\ \emptyset, & \textit{otherwise} \end{cases}$$

With the help of the techniques presented in this section, it is possible to define a consistent feature mapping for artifact elements of a product line based on existing variability documentations in the form of family models. The mapping from Definition 4.13 can be used during the application of the methodology presented in [162].

4.5 Evaluation

During the *SPES_XT* project a multitude of external and internal case studies have been conducted by the project partners. Internal case studies were conducted by industrial partners on real-world software artifacts using the methods developed during the runtime of the project, while external case studies were developed by an industrial partner and provided for project-wide utilization. These external case studies included a set of artifacts from a fictional product line based on a real-world example. One of these case studies was the 'automotive case study' developed by Daimler [111, p. 11-25]. In this case study, the development artifacts of two product lines were provided to the other project partners. One of these systems is a control system for the external light of an automotive (EL) while the other one is a control system responsible for the velocity of a driver assistance system (DAS).

We evaluated the methodology presented in this chapter with the artifacts related to the DAS system:

- **Functionmodel** $\mathcal{M}_{FAS}$**:** MATLAB/Simulink model (FAS model introduced in Section 3.5.1)
- **Requirements documentation** A_R**:** Formal module from IBM Rational DOORS
- **Test case description** A_T**:** Formal module in IBM Rational DOORS
- **Feature, family and variant description models** $F, F(A_R), F(A_T), V$**:** Created with pure::variants

Test cases that realize specific requirements are linked to them via cross module links. For each DOORS module, a family model is available that maps features to their respective DoorsObjects via their unique identifier. Further links exist between feature, family and variant description models, expressed by the unique identifier in the respective model, e.g. a link to a feature from a component in a family model is captured by storing the unique id of the feature in the feature model.

While the artifacts of the DAS system further include the functionmodel and variant description model, we could only apply the automatic feature derivation procedure and the consistency checker to the requirements documentation, test specification and their corresponding family models. Only these artifacts contained sufficient traceability links to apply the processes described in Section 4.3 and 4.4.

Both processes have been implemented as an extension component in the artshop framework. Based on the information provided by the tool adapter, intra- and inter-artifact traceability links can be reconstructed and stored in the model repository. These links can then again be queried and validated by the consistency checker.

The next section describes how we applied the feature mapping derivation procedure to the provided artifacts.

Artifact	Elements	Links				
		A_R	A_T	$F(A_R)$	$F(A_T)$	F
A_R	86	-	52	86	-	176*
A_T	136	52	-	-	136	128*
$F(A_R)$	86	86	-	-	-	86
$F(A_T)$	136	-	136	-	-	101
F	31	176*	128*	86	101	-

Table 4.1: Overview of the artifact elements in the DAS system including imported and computed links(*)

4.5.1 Execution of the Derivation Procedure

We first imported all artifacts into the artshop framework and stored them in the repository. An overview of the amount of involved artifact elements is shown in Table 4.1.

After that, we resolved the links between the requirements documentation A_R and the test case specification module A_T by querying the repository for the unresolved association ends of imported ExternalDoorsLinks. As each unresolved link is imported for both modules, we only resolve one of them and delete the redundant one. This results in 52 links shared across both modules. The tool adapter for feature and family models automatically resolves inter-dependencies between these artifacts; therefore, no further adaptations were necessary. In total 187 links exists between the feature and family models. Traceability links between family model and DOORS modules can then be recreated based on the information stored in the family model by mapping each component to its corresponding DoorsObject, resulting in 222 traceability links between the family models and their associated DOORS modules. Finally, features can be mapped to their linked artifact elements contained in the formal modules by transitively following the traceability links via the family model as described in Definition 4.13, resulting in 176 features being mapped to the elements of A_R and 128 features being mapped to the elements of A_T.

The complete link derivation process takes approximately 330 ms, including all database queries needed for the resolution process. This represents a significant speed up in comparison to a manual feature annotation process, as manual annotation of features to their respective development artifacts in *artshop* takes around 5-10 minutes for each individual feature, as all artifacts have to be thoroughly examined to capture all feature dependencies. As the artifacts of the DAS case study are relatively small, it can be assumed that even more time is needed to create a feature mapping for industrial-size artifacts manually. In contrast, the automatic derivation procedure can automatically extract and map all features defined in the variability documentation to the respective development artifacts, based on the assumption that the variability documentation contains the correct mapping.

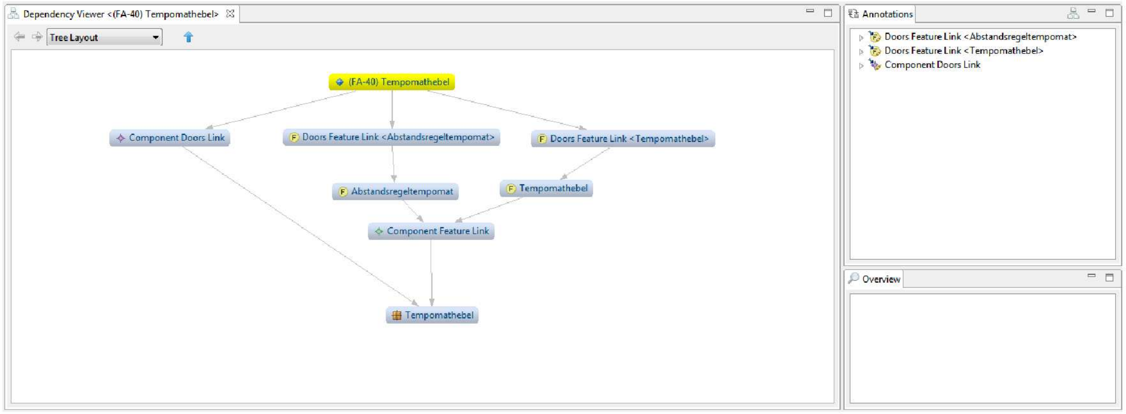


Figure 4.7: Dependence view showing the traceability links of a requirement in artshop

4.5.2 Execution of the Consistency Check

After a feature mapping has been obtained, its consistency can be checked using the technique described in Section 4.4. On average, the consistency check took around 2.6 ms on the artifacts of the DAS system, while assuming the feature mappings of $F(A_T)$ as a reference for the consistency check. Overall, 37 inconsistencies by incompleteness have been found between the variability documentation of requirements and test case specification. Of these 37 inconsistencies, 27 could be mapped to an erroneous feature mapping on the root level of one of the analyzed family models, resulting in an inconsistent set of obligatory features for the components of this family model. 8 other inconsistencies can be attributed to the application of different mapping paradigms within the feature mappings of both family models. While $F(A_R)$ also maps features to DoorsObjects representing intermediate hierarchy layers that group a set of DoorsObjects to a certain feature, $F(A_T)$ only contains feature mappings to the leaf nodes of the associate DOORS module A_T. The other two inconsistencies are attributed to two individually incomplete mappings between two pairs of DoorsObjects.

The found inconsistencies could easily be resolved using the descriptions of the individual inconsistencies.

4.6 Conclusion and Future Work

In this chapter, we have presented a method for the automatic derivation of feature mappings from the variability documentation of a software product line managed by pure::variants. Feature mappings are derived via the relationships stored in feature and family models and mapped to the development artifacts references by the family models. The correctness of the extracted mappings can be verified via an inter-artifact analysis using traceability links between elements of artifacts that were part of the feature mapping derivation process.

We empirically evaluated the approach on the artifacts of a case study from the SPES_XT project. The feature derivation procedure can significantly speed up the enrichment of artifacts with variability information, in comparison to a manual feature annotation approach. Moreover, we checked the consistency of the extracted feature mappings using the proposed consistency checker and were able to find a set of inconsistent feature mappings in the variability documentation. While the proposed approach performed well on the artifacts of the FAS case study, further evaluations of the techniques should be conducted on industrial sized artifacts.

The techniques proposed in this chapter are well suited as a pre-processing step of other analysis techniques such as the consistency analysis we described in [162]. In addition, the extracted information could potentially be used for the creation of context-sensitive views, by only showing artifacts connected to the features selected in a variant description model. The artshop framework already supports the visualization of the traceability links computed by the proposed approach as it can be seen in Figure 4.7. By double clicking, the links of the selected model element are also propagated into this view, allowing the navigation of extracted traceability information.

Limitations of the current approach still exist with respect to the resolution of inconsistencies found by the consistency checker. Currently, inconsistencies need to be manually resolved in the source artifacts that were imported into artshop. It would be desirable to employ an automatic resolution mechanism that can directly correct found inconsistencies into the source artifacts of pure::variants.

5 Dependency Analysis and Slicing of MATLAB/Simulink Models

MATLAB/Simulink was already introduced in Section 2.2 as an important development tool for the creation of model-based software in the automotive domain. Industrial sized MATLAB/Simulink models may contain dozen of hierarchy levels and tens of thousands of blocks due to the sheer amount of functionality modeled within these diagrams. MATLAB/Simulink offers certain architectural patterns to cope with the complexity introduced with such an amount of elements. These include Subsystem blocks needed to modularize a model and encapsulate distinct functions with a defined interface. Bus signals can be used to compose multiple visual signals into one composite signal. Both of these patterns reduce the overall visual complexity of a model but also hide information from the modeler that might be needed to understand the model, i.e. dependencies between model parts. As bus signals containing more than 100 signals that are routed across multiple hierarchy levels are common in industrial size models, automatic techniques are needed to cope with the complexity of manual dependency review. These techniques need to be able to reconstruct the signal flow in a MATLAB/Simulink model, while at the same time respecting all factors influencing dependencies within a model, e.g. block and signal flow types as introduced in Section 2.2.1.

Dependency analysis techniques have multiple applications during the design and maintenance of model-based software created with MATLAB/Simulink. They can be leveraged to compute context-sensitive views including only parts of the model that are currently relevant for the modeler during model maintenance, testing and debugging. Such model slices could also be used to reuse a subset of a given model with respect to a certain functionality by exporting the slice of the model into a separate model. Further application scenarios include change-impact analyses during model evolution that can be leveraged to realize incremental static analyses.

5.1 Overview

In this chapter, we first present a dependency analysis for MATLAB/Simulink models to determine signal dependencies between blocks within a model. The analysis distinguishes between data and control dependencies, which are incorporated into a dependence graph of the Simulink model, similar to the concept of Program Dependence Graphs first introduced by Ferrante [52]. Besides dependence relations from related work on dependency analysis of MATLAB/Simulink models [9, 97, 115, 116], the presented dependency analysis respects the influence of all block and signal flow types on data dependence that have been introduced in Section 2.2.1. Moreover, we propose the extraction of control dependency relations directly from MATLAB/Simulink.

To leverage the results of the dependency analysis for complexity reduction of a model with respect to a scope of elements, we integrate the results of the analysis into a slicing algorithm adapted to MATLAB/Simulink models. In later chapters of this thesis, the dependency graph and slicing algorithm will be used to realize further structural static analyses as they provide powerful tools to reason about the internal structure of a MATLAB/Simulink model, which was also noted by Hu et al. in [71]. We evaluate the resulting approach against state-of-the-art slicing techniques from the literature using models from industrial case studies, academic projects as well as models provided as examples by MATLAB/Simulink. Finally, we propose a definition for data dependence in MATLAB/Stateflow models and discuss how this dependency relation can be incorporated into the presented slicing algorithm.

5.1.1 Related Work

Merschen et al. [97] present a signal reconstruction approach for MATLAB/Simulink models, which respects implicit signal flow via Goto and From blocks. The presented analysis is based on a model created by the tool adapter that has been mentioned in Section 3.1.2 and includes virtual lines across subsystem borders and between Goto and From blocks to enable the seamless traversal of the model. Signals are identified and created based on labels attached to lines, which might lead to an inaccurate representation of the signal flow graph when bus-capable blocks are used in the model. The authors do not address implicit signal flow via datastore blocks, signal renaming and only partially cover tracing atomic signals through bus signals.

Bender et al. [9], introduce an approach to analyze the signal flow within Simulink models. In their work, explicit and implicit interfaces of subsystems are defined and discussed. The authors extensively explain implicit signal flow between Goto/From and Datastore blocks, but omit the discussion of signal flow across bus signals for matters of simplicity. The influence of specific block types on data dependence is not mentioned as part of their work.

The first slicing approach for MATLAB/Simulink models has been presented by Reicherdt et al. in [116], which is based on program dependence graphs. The authors construct a data dependence relation only with regard to the lines that are used to connect blocks. This disregards the influence of virtual blocks and bus-capable blocks onto the signal flow, potentially leading to imprecise slices if bus signals are used within an analyzed model. The authors further present an algorithm to compute conditional execution contexts needed to create the control dependence relations in a Simulink model. In contrast, the slicing approach introduced in this thesis uses information available within Simulink to compute the control dependence relation. In his PhD thesis, Reicherdt extended the data dependence relation of his approach from [116] to bus signals [115]. While signal information is computed in a different way, the approach is similar to the one used in this thesis, but does not include indirect signal flow across DataStoreRead and DataStoreWrite blocks obtained by the sorted order of the model. In addition, Reicherdt ignores signal boundaries imposed by nonvirtual blocks, e.g. an atomic signal exchanged between two nonvirtual blocks is not propagated over the receiving nonvirtual block. These boundaries are not relevant for the computation of slices but need to be considered during the model smell analyses presented in Chapter 7. In addition, we provide an over-approximation for the input-output relationships of MATLAB/Stateflow charts (see Section 5.6). Reicherdt uses computed signal information during the verification of MATLAB/Simulink models using the Boogie framework [117]. Similar information are used in the static value analysis framework developed by Dernehl et al. [40, 41, 42, 43, 44], that uses the signal information of models imported with the artshop framework.

Another slicing algorithm was presented by Pantelic et al. [108] as part of the *Reach/-Coreach* tool. Here the definitions for dataflow from [9] are used to perform impact analysis and slicing on the model. Again, the characteristics of bus signals and the impact of specific block types on the dataflow properties of the model are not discussed within this work. Neither the approach of Reicherdt et al. nor Pantelic et al. discuss dependency analysis of MATLAB/Stateflow models, which is also addressed in this chapter.

In addition to the presented approaches for slicing Simulink models, slicing algorithms have been applied to various kinds of models, e.g. *extended finite state machines*, *Statecharts* and *UML/SysML* models.

An extensive survey of slicing techniques for state-based models, i.e. extended finite state machines and statecharts, is given in [5]. Lano et al. [89] introduce slicing techniques for a subset of the UML, i.e. class diagrams, state machines and communicating sets of state machines. Their approach respects both declarative elements of the UML such as pre- and post-conditions and imperative elements (state machines).

Nejati et al. [103] propose to use slicing during safety certification by extracting model fragments from SysML models relevant to a particular safety requirement. The authors use traceability links as a starting point for their slicing algorithm and then compute a set of relevant elements in all block definition, internal block and activity diagrams.

Lallchandani et al. present a slicing approach for UML models via the construction of a so-called Model Dependence Graph based on the dependence relations of the model and slice this graph to obtain the actual model slice [88]. A similar approach is used within this thesis to realize a slicing algorithm on MATLAB/Simulink models.

5.1.2 Contributions and Bibliographic Notes

In this chapter, we provide an extensive discussion of factors influencing the signal flow in MATLAB/Simulink models and propose an algorithm to reconstruct all atomic and bus signals exchanged within these models. We use the signal information derived by this algorithm, to define a signal flow-based data dependence relation and use this relation in conjunction with further control dependence information extracted from MATLAB/Simulink for the computation of the dependence graph of a model. To our knowledge, this is the first approach to include all factors influencing data dependencies in MATLAB/Simulink models. Finally, we propose a slicing algorithm for MATLAB/Simulink models by performing a reachability analysis on the computed dependence graph. Fur-

```
1  read(n);
2  i = 0;
3  sum = 0;
4  product = 1;
5  while (i < n)
6    sum = sum + i;
7    product = product * i;
8    i = i + 1;
9  write(sum);
10 write(product);
11
```

(a)

```
1  read(n);
2  i = 0;
3  sum = 0;
4
5  while (i < n)
6    sum = sum + i;
7
8    i = i + 1;
9  write(sum);
10 write(product);
11
```

(b)

Figure 5.1: Example program (a) and slice of (a) with slicing criterion <9, sum> (b)

ther, we provide a proof-of-concept algorithm to extend the dependence analysis of the algorithm to Chart blocks from MATLAB/Stateflow.

The proposed techniques are extensively evaluated and their importance for the analysis of MATLAB/Simulink models is highlighted by their use in further application scenarios.

The flow-based definition for data dependence and the slicing algorithm have partially been published in [59].

5.2 Foundations

We will first introduce basic concepts and terminology related to dependency analysis and slicing on program code before we map these concepts to MATLAB/Simulink models.

5.2.1 Dependency Analysis

The literature distinguishes two types of dependencies between statements of a program: data and control dependence. They are defined based on the control flow graph (CFG) of a program P. Such a graph contains a node for each program statement in P and an edge between two nodes representing a flow of control between two statements.

Definition 5.1 (Control Flow Graph [14])**.**
A control flow graph for a program P is a graph $G_{CFG} = (N, E)$ in which each node is associated with a statement from P and the edges represent the flow of control in P. Let V be the set of variables in P. With each node $n \in N$ (i.e., each statement in the program and node in the graph) associate two sets: $REF(n)$, the set of variables whose values are referenced at n, and $DEF(n)$, the set of variables whose values are defined at n. $START$ and $STOP$ nodes represent the start and end of the respective program P within G_{CFG}.

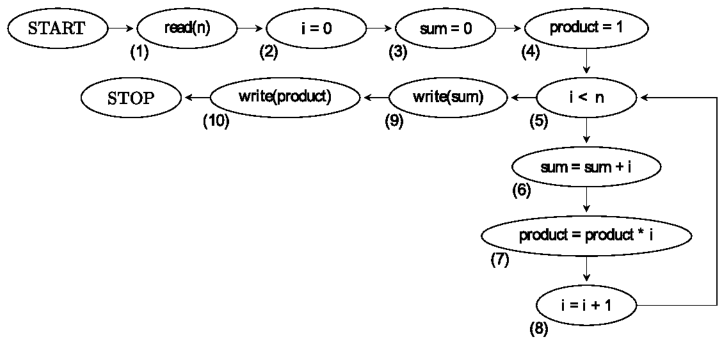


Figure 5.2: CFG of the program displayed in Figure 5.1a adapted from [151]

Data dependencies between nodes of a CFG are intuitively formed based on so-called def-ref relationships between program statements, i.e. if a variable is defined at node i and later referenced at node j.

Definition 5.2 (Data Dependence [151]).
A node j is data-dependent on a node i via a variable x if the following conditions hold:

- $x \in DEF(i)$
- $x \in REF(j)$
- *There exists a path between node i and j without intervening definitions of x*

Control dependence is a property of control flow between nodes in a CFG and defined in terms of post-dominance of nodes. A node i in the CFG is post-dominated by another node j, if all paths from node i to the STOP node contain node j [151].

Definition 5.3 (Control Dependence [52]).
A node j is control-dependent on a node i if the following conditions hold:

- *There exist a path p from node i to node j so that node j post-dominates every node on p excluding node i and j*
- *Node i is not post-dominated by node j*

Based on the CFG and the two definitions for data and control dependence, dependency analysis can be performed on a program. Consider the CFG shown in Figure 5.2, which represents the control flow of the example program in Figure 5.1a. The example program reads an input into variable n and, based on its value, computes the sum and product of all natural numbers up to n within the variables `sum` and `product` and outputs their values at the end of the program. In terms of data dependence, node 5 in the CFG is data-dependent on node 2, as $i \in DEF(2)$ and $i \in REF(5)$ and no intervening definition of variable i is present on the path between node 2 and 5. Furthermore, node 8 is control dependent on node 5 as node 8 post-dominates all nodes between node 5 and 8 and node 5 is not post-dominated by node 8.

5.2.2 Slicing

Program slicing, as introduced by Weiser [160], is a mechanism to extract a *slice* of a program that affects a given point of interest [151]. Formally, such a point of interest encompasses one statement s of a program and a set of variables V and is defined as the *slicing criterion* $< s, V >$ of a given slice. Based on the dependence relations discovered by dependence analysis, a slice can be computed by transitively determining all program statements that affect the values of the variables from V at program point s. An example slice is shown in Figure 5.1b. Here, the program shown in Figure 5.1a is sliced with respect to the slicing criterion $<s_9,$`sum`$>$, with s_9 being the program statement of line 9, i.e. all statements that do not affect the value of variable `sum` in line 9 are removed.

While the original approach introduced by Weiser [160] utilized data-flow equations to calculate a slice from a program's CFG, most modern slicing techniques utilize *Program Dependence Graphs (PDG)* for slice computation. As the CFG, the PDG is a directed graph that contains a vertex for each statement of a program P. In contrast to CFGs, the vertices of a PDG are connected based on their dependencies to each other, which can be discovered by dependency analysis on the CFG.

Definition 5.4 (Program Dependence Graph [52])**.**
A program dependence graph for a program P is a directed graph $G_{PDG} = (V, E)$ with the following properties:

- *N: Contains a vertex for each statement in P and an entry vertex representing the start of the program.*
- *E: Contains a set of pairs of $V \times V$, representing data and control dependence edges between the vertices of V.*

The entry vertex cannot be the target of any dependence edge contained in E and all other vertices in V are directly or transitively control dependent on the entry vertex.

Using PDGs, program slices can be calculated by solving a vertex reachability problem on the PDG [107]. For a given vertex v in the PDG, it is sufficient to compute all vertices that v directly or transitively depends on. Therefore, the slicing criterion is no longer a combination of a statement and a set of variables but is expressed by the statement associated to the vertex in the PDG and all variables that are defined/used within this statement.

Figure 5.3 shows the PDG for the example program from Figure 5.1a. Here, solid edges represent control dependence between two vertices, while dashed edges represent data dependence. All grayed out nodes are directly or transitively dependent on the node with the label *write(sum)* corresponding to statement 9 in the example program shown in Figure 5.1a. These vertices correspond to the slice with the slicing criterion $<s_9,$ `sum`$>$ shown in Figure 5.1a.

Slicing algorithms may be classified based on various properties as highlighted by Silva in [144] and Tip [151]. In the context of this thesis, we only consider a subset of these classifications that have first been mentioned by Venkatesh in [155].

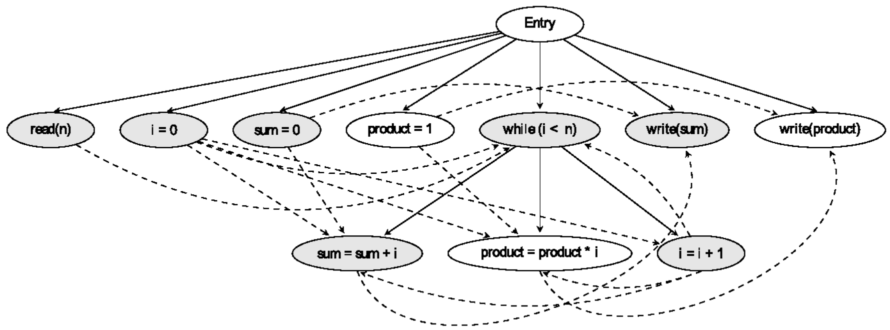


Figure 5.3: PDG of the example program from Figure 5.1a adapted from [151]

- **Slice direction**: A slice can be calculated either in forward or backward direction. Up until now, we only considered backward slices that contain all program statements that influence the program location given by the slicing criterion. Forward slices contain the statements that depend on the statement described by the slicing criterion.
- **Static vs. dynamic slicing**: A static slicing technique does not make any assumptions of the state of the variables of the program and therefore contains all possible execution paths from or to the respective slicing criterion. By utilizing information about a given state of the program, a dynamic slicing technique can calculate a slice on a program that is actually reachable based on a given state of a program. Such techniques require the evaluation of conditional statements to compute control and data dependence relations correctly.
- **Closure vs. executable slice**: A slice can contain either an executable subset of the statements of the program or a non-executable subset of the program through a closure of dependencies.

5.3 Signal-Flow in MATLAB/Simulink

In comparison to program code, where data dependencies between statements are expressed via def-ref relationships of variables declared/used within these statements (see Definition 5.18), data flow in MATLAB/Simulink is more or less directly modeled by the lines connecting the blocks in a model. Lines carry one or multiple signals from one outport to one or multiple inports attached to the blocks of a model (see Section 2.2.1). Signals are generated by the nonvirtual blocks of the model. While the general data flow can be visually followed by tracing the lines connecting blocks in the model, the application of architectural pattern such as bus signals allow the recursive composition of multiple signals into one visual line. Moreover, some blocks exchange signals via indirect signal flow as introduced in Section 2.2.1. The usage of these architectural patterns as well as the usage of indirect signal flow within a model reduce the visual complexity of

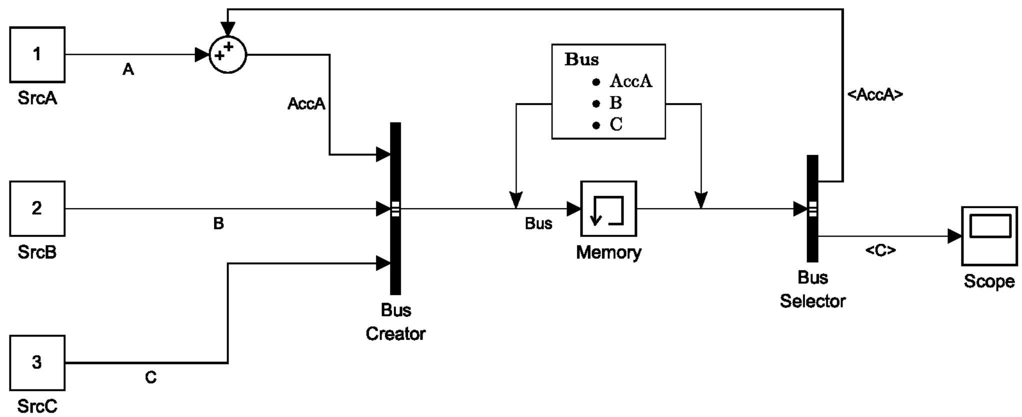


Figure 5.4: Signal propagation over different block types

the model but complicate the visual analysis of dependency relations between blocks of the model.

Dependencies between two blocks b_1, b_2 are established based on the signals b_1 emits and b_2 consumes. The type of a block, nonvirtual or virtual, determines if it emits new signals, consumes signals or just propagates them. Nonvirtual blocks emit/consume signals and realize the semantic behavior of a model. Virtual blocks are the organizational elements of a model and simply propagate received signals. A block can be either virtual or nonvirtual but a subset of virtual blocks becomes nonvirtual under certain conditions, e.g. their configuration or their immediate environment. Another block type are bus-capable blocks, which are capable of emitting and receiving bus signals (see Section 2.2.1). If a bus signal is propagated over a bus-capable block, it is continued with the same signal characteristics, i.e. bus structure, signal names etc. All virtual blocks are also bus-capable blocks, with a subset of bus-capable blocks being nonvirtual. While nonvirtual bus-capable blocks consume an incoming bus signal, they also emit a new bus signal, which is a copy of the signal they have received. Tracing an atomic signal s contained within a bus signal that is propagated over a nonvirtual bus-capable block b_{bcb} requires a mapping that maps the atomic signal received by b_{bcb} to its equivalent output signal. Otherwise, the complete bus signal emitted by the bus-capable block needs to be traced, which does not solely depend on the initially received signal s. Consider signal C emitted from the Constant block *SrcC* in Figure 5.4. The signal is entered into the bus signal *Bus* created by the BusCreator block and ends at the Memory block. A copy of the bus signal is emitted by the Memory block leading to the termination of the copy of signal C at the Terminator block. In terms of data dependence, mapping the signal C that terminates at the Memory block to its equivalent copy in the emitted signal is therefore preferred as it results in a finer level of signal traceability.

In the remainder of this section, we will provide a flow-based definition for data dependence in MATLAB/Simulink models and discuss how control dependencies are formed based on the sorted order of a model. The flow-based definition of data dependence accounts for the impact of the block types on signal propagation characteristics described in the previous paragraphs. Indirect signal flow across model hierarchy levels and Goto/From blocks is reconstructed using the concept of virtual lines introduced in Section 3.3.1, while signals exchanged via DataStore blocks are handled separately.

To reason about the signal flow in a MATLAB/Simulink model, we first define a representation of signals within a Simulink model, to express the possibility of a signal being propagated over a multitude of virtual blocks.

5.3.1 Signals in MATLAB/Simulink

Signals can be propagated over multiple lines and blocks depending on the block type of the receiving blocks. Such a signal can be partitioned into multiple segments, with each segment corresponding to the signal flow of the signal between an outport and an inport. In the following, we will use the formalization and definitions introduced in Section 2.2.4 as the basis for the definition of signal flow in MATLAB/Simulink model $\mathcal{M}$. We start with the definition of the signal flow between two ports called a signal segment.

Definition 5.5 (Signal Segment)**.**
A signal segment c contained in a MATLAB/Simulink model $\mathcal{M}$ is defined as a triple $c = (L_s, L_d, l)$ with:

- *Source location of segment $L_s = (b_s, p_s)$ with $b_s \in B$ and $p_s \in ports_{out}(b_s)$*
- *Destination location of segment $L_d = (b_d, p_d)$ with $b_d \in B$ and $p_d \in ports_{in}(b_d)$*
- *$l =$ The fully qualified signal name of c*
- *$(p_s, p_d) \in L$ or $p_s = p_d = \perp$*

The set of signal segments of a MATLAB/Simulink model $\mathcal{M}$ is denoted as C.

A signal segment $((b_1, p_{out}), (b_2, p_{in}), l)$ describes a signal exchanged between two blocks b_1 and b_2 that are connected via their respective in- and outport p_{in}, p_{out}. The relationship between an outport and an inport of two blocks is typically established based on a line $(p_{out}, p_{in}) \in L$ that connects these ports. If a signal segment represents a signal exchanged via indirect signal flow, the source and destination ports may be set to $\perp$ as some of these blocks do not have ports emitting and receiving the indirectly exchanged signal, e.g. DataStore blocks.

The current fully qualified signal name of a signal segment is stored in l. A signal segment not contained in a bus signal is usually named after the line $e = (p_{out}, p_{in})$ it is associated with or, in case of indirect signal flow, by the last line the signal was propagated over. To ease later definitions, we assume that each line is associated with a name. If a signal segment is contained in a bus signal, its fully qualified signal name is constructed as a sequence of names derived from the bus signals it is contained in, in descending hierarchical order.

Definition 5.6 (Fully Qualified Signal Name (FQSN)).
Let I be the finite set of all signal names used in a Simulink model $\mathcal{M}$.

- *The structure of the $FQSN(c)$ of a signal segment c is defined as a sequence q over I where q is an ordered list $\langle q_1...q_n \rangle$ with q_i $1 \leq i \leq n$ denoting a signal name from I and n the length of the sequence*
- *The empty sequence is defined as $\langle\rangle$ and has a length of 0*
- *A sub sequence $q|^i$ of a sequence q with length n is defined as $q|^i = \langle q_i...q_n \rangle$ for $i < n$ and as $\langle\rangle$ otherwise*
- *The concatenation operation $\circ$ of two sequences $q = \langle q_1, ..., q_n \rangle$ and $v = \langle v_1, ..., v_m \rangle$ over I is defined as*
$$l \circ v = \langle q_1...q_n v_1...v_n \rangle$$

$FQSN(c)$ describes the hierarchical sequence of bus signals, a signal segment c is located in, based on their respective signal names. Reconsider the example model shown in Figure 5.4. The fully-qualified signal name of the signal segment c of signal C between the BusCreator and Memory block would be $FQSN(c) = \langle Bus, C \rangle$. The first element in the sequence of names represents the name of bus signal *Bus*, expressing the inclusion of c in a signal segment of this signal. The last element represents the current name of the signal represented by c.

Based on the definition of $FQSN(c)$, we can formulate a signal inclusion relationship based on two signal segments.

Definition 5.7 (Signal Containment).
A signal segment $c = (L_s, L_d, FQSN(c) = \langle c_1...c_m \rangle)$ is contained in another signal segment $d = (L_s, L_d, FQSN(d) = \langle d_1...d_n \rangle)$ iff it holds that $\forall\ i \in [1..n], c_i = d_i$. The signal segment containment relation shall be defined as

$$\subset_c : C \rightarrow \mathbb{B}$$

As mentioned before, based on the block type of the block b_d a signal segment $c = ((b_s, p_s), (b_d, p_d), l)$ ends at, the signal s represented by c might be propagated over b_d without being changed. To represent a continuation of s from b_d, a succeeding signal segment $d = ((b_d, p'_s), (b'_d, p'_d), l')$ can be created that starts at one of the outports of b_d. Multiple succeeding signal of a signal s can then be aggregated into a signal path through a model.

Definition 5.8 (Signal Path).
A signal path s^p=(L_s,L_d,C_p) describing the trace of a signal through a model $\mathcal{M}$ is defined as:

- *Signal path start location $L_s = (b_s, p_s)$ with $b_s \in B$ and $p_s \in ports_{out}(b_s)$*
- *Signal path destination $L_d = (b_d, p_d)$ with $b_d \in B$ and $p_d \in ports_{in}(b_d)$*

Table 5.1: Functions for the navigation and exploration of the signal segments C, paths S^P and signals S of a MATLAB/Simulink model $\mathcal{M} = (B, P = P_{in} \cup P_{out}, L, I, f_p, h)$

Relation	Logical signature	Definition
Signal segments of signal path	$seg_{S^P} : S^P \to \mathcal{P}(C)$	$seg_{S^P}((L_s, L_d, \langle c_1 ... c_n \rangle)) = \{c_1, .., c_n\}$
Signal segments of signal	$seg_S : S \to \mathcal{P}(C)$	$seg_S((L_s, P_S)) = \bigcup_{p \in P_S} seg_{S^P}(p)$
Signal segment at port	$seg_P : P \to \mathcal{P}(C)$	$seg_P(p) = \{c \mid c \in C \wedge (p_1 = p \vee p_2 = p)\}$ with $c = ((b_1, p_1), (b_2, p_2), l)$
Signal paths containing segment	$paths : C \to \mathcal{P}(S^P)$	$paths(c) = \{s^p = (L_s, L_d, \langle c_1 ... c_n \rangle) \mid \exists\, i \in \mathbb{N} : i \leq n \wedge c_i = c\}$
Signal for segment	$signal_c : C \to S$	$signal_c(c) = s = (L_s, P_s)$ so that holds: $path(c) \subseteq P_s$
Destination of signal path	$dest_{S^P} : S^P \to P$	$dest_{S^P}((L_s, (b_{dest}, p_{dest}), C_P)) = p_{dest}$
Destinations of signal	$dest_S : S \to \mathcal{P}(P)$	$dest_S((L_s, P_S)) = \bigcup_{p \in P_S} \{dest_{S^P}(p)\}$

- *Sequence of signal segments* $C_p = \langle c_1 ... c_n \rangle$ *with* $c_i = ((b_{s_i}, p_{s_i}), (b_{d_i}, p_{d_i}), l_i)$*:*
  - $p_{s_1} = p_s$ *and* $p_{d_n} = p_d$
  - $b_{d_i} = b_{s_{i+1}} : \forall\ 1 \leq i < n$

The set of signal paths in a MATLAB/Simulink $\mathcal{M}$ *is denoted as* S^P*.*

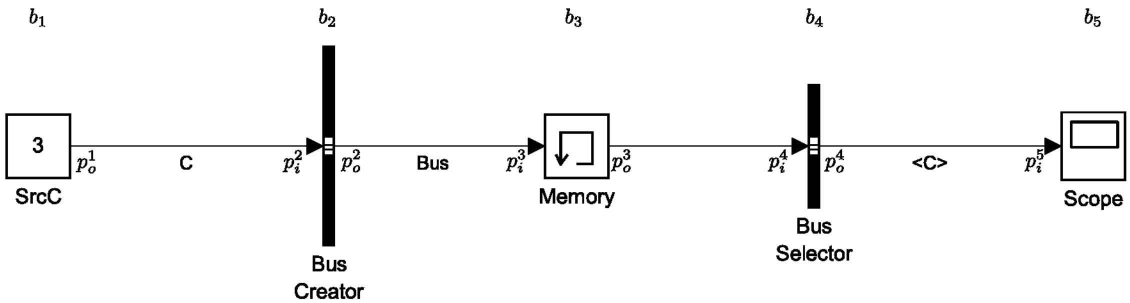
A signal path s^p starting at $L_s = (b_s, p_s)$ describes a trace of a signal through the model that is created at L_s and terminates at $L_d = (b_d, p_d)$. The segments contained within the sequence C_p must succeed each other, i.e. the start and destination blocks of subsequent signal segments in C_p are equal. Thus, a signal path describes exactly one path a signal takes through the model, without being changed, i.e. all blocks on the path must either be virtual or bus-capable blocks.

By applying both definition 5.5 and 5.8, we can now define a signal emitted from the port of a specific block.

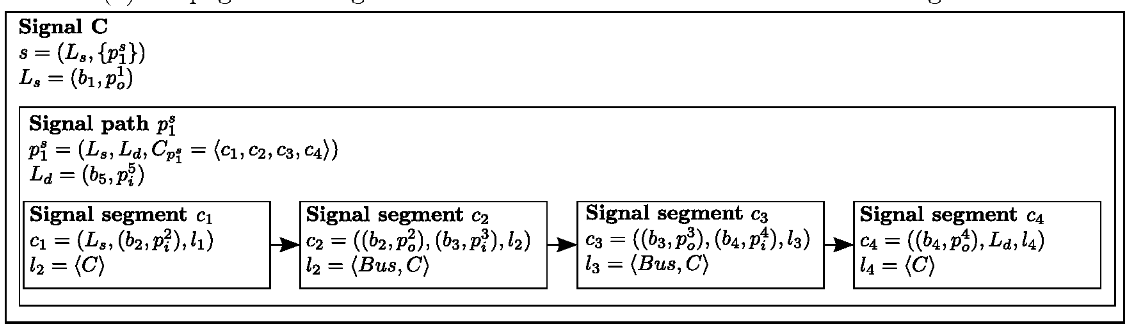
Definition 5.9 (Signal)**.**
A signal s *is defined as* $s = (L_s, P_s)$ *and consists of*

- *Signal start location* $L_s = (b_s, p_s)$ *with* $b_s \in B$ *and* $p_s \in ports_{out}(b_s)$
- *Set of signal paths* P_s *for that holds:* $\forall\ p_S = (L'_s, L'_d, C') \in P_s : L_s = L'_s$

The set of signals of a MATLAB/Simulink model $\mathcal{M}$ *is denoted as* S*.*



(a) Propagation of signal C over the blocks of the model shown in Figure 5.4



(b) Signal representation of signal C in the model excerpt from Figure 5.5a

Figure 5.5: Application of signal representation on signal C from shown in Figure 5.4

A signal emitted from the port of a block bundles all signal paths that start at that port and therefore contains information about all blocks that are potentially dependent on the signal. With the help of this definition, we can now formally represent the signal flow relationships of signals emitted and received by blocks of a Simulink model.

An example for a signal with one signal path would be signal C emitted by the block SrcC in Figure 5.4. If traced over the nonvirtual bus-capable Memory block, the signal path contains four signal segments of which two are contained in a bus signal. The blocks reachable by this signal are visualized in Figure 5.5a. By using Definition 5.9, 5.8 and 5.5 we can represent signal C as shown in Figure 5.5b.

In addition, we provide a set of functions that ease the formalization of latter definitions in Table 5.1. These functions enable the extraction of properties out of signals and signal paths, enable the ascending navigation between the different granularity levels of signals and provide a mapping for signal segments flowing over a specific port p.

5.3.2 Data Dependence in MATLAB/Simulink

As we are now able to represent the flow of signals within a MATLAB/Simulink model $\mathcal{M}$, we can describe how signals are propagated. Propagation will be described in the form of expansion functions, which compute a set of succeeding signal segments based on a given signal segment c. Based on the block type c ends at, an appropriate expansion function must be chosen that respects the signal propagation characteristics introduced at the beginning of Section 5.3. Starting from the set of initial signal segments of a signal s created by a nonvirtual block, these functions can then iteratively be applied to construct the signal paths of s. Expansion functions receive a signal segment c as input and create a set of succeeding signal segments. If a function determines that a signal terminates at a block, the function returns $\emptyset$.

Definition 5.10 (Nonvirtual Block Expansion Function)**.**
The expansion function for nonvirtual blocks (NVB) is defined as

$$expand_{NVB}(c) = \emptyset$$

Each signal received by a nonvirtual block terminates, as a nonvirtual block uses its received signals to calculate a new set of output signals. Therefore, $expand_{NVB}$ always returns an empty set of signal segments. An example for this behavior is the signal $AccA$ that is received by the Sum block in Figure 5.4, which terminates at the Sum block as its value is used to compute the output value of the block.

A block type that supports the propagation of input signals are virtual blocks.

Definition 5.11 (Virtual Block Expansion Function)**.**
The expansion rule for virtual blocks (VB)

$$expand_{VB}(c = (L_s, L_d = (b, p), l))$$

is defined as

$$\begin{aligned} expand_{VB}(c) &= \{c' | succ_c(c, c')\} \\ with\ c' &= ((b, p_s), (b', p_d), h((p_s, p_d)) \circ l|^2)\ and \\ succ_c(c, c') &= ((f_p(p_s) = f_p(p_d)) \wedge (f_p(p_d) = b') \wedge \\ &\quad (p_s, p_d) \in L) \end{aligned}$$

Virtual blocks do not affect the semantic of a model, which means that the function $expand_{VB}$ will expand every segment over the current block b, as long as it has a successor b'. When a segment is expanded, the first name in the sequence of $FQSN(s')$ is changed according to the line label mapped to (p_o, p_i), while the rest of the FQSN remains unchanged. This way, signals contained in a bus signal are not renamed when the name of their containing signal changes. Only the first qualifier of their FQSN is updated.

In Section 2.2.1, we have introduced the BusCreator, BusSelector and BusAssignment blocks as a way to create, resolve and manipulate bus signals. Even though these blocks also belong to the block category of virtual blocks, they present exceptions to the $expand_{NVB}$ rule and need to be handled separately. In the following, we will detail how the creation, resolution and manipulation of bus signals during signal propagation is handled.

Definition 5.12 (BusCreator Block Expansion Function)**.**
Let $unique_b : I \rightarrow I$ be the aliasing function of BusCreators described in Section 2.2.1 for a BusCreator block b. The expansion function for BusCreator blocks (BC)

$$expand_{BC}(c = (L_s, L_d = (b, p), l = \langle l_1 ... l_n \rangle))$$

is defined as

$$\begin{aligned} expand_{BC}(c) &= \{c' | succ_c(c, c')\} \\ \textit{with } c' &= ((b, p_s), (b', p_d), l') \\ \textit{and } l' &= h((p_s, p_d)) \circ unique_b(l_1) \circ l|^2 \end{aligned}$$

To reflect that the signal represented by the expanded segment c' is contained within the bus signal emitted by *BusCreator* b, the FQSN of the initial segment c is prepended with the name of this signal and set as $FQSN(c')$. Additionally, the aliasing function $unique_b$ is applied to the first segment of $FQSN(c)$ to assert that there exists no signal in the bus signal that has the same name as another signal. As the BusSelector and BusAssignment blocks resolve and manipulate signals in a bus signal via their FQSN, we need to assure that the FQSN of all signals within a bus signal are unique at all times.

To extract a signal segment from a bus signal, it needs to be expanded over a *BusSelector* that selects the signal represented by this segment.

Definition 5.13 (BusSelector Block Expansion Function)**.**
Let $I_{BS}(b)$ be the set of tuples of fully qualified signal names of the signals and their resulting ports selected by a BusSelector block b. The expansion function for BusSelector blocks (BS)

$$expand_{BS}(c = (L_s, L_d = (b, p), l = l_{pre} \circ l_a))$$

is defined as

$$\begin{aligned} expand_{BS}(c) &= \{c' | \\ succ_c(c, c') \wedge (l_{pre}, p_s) &\in I_{BS}(b)\} \\ \textit{with } c' &= ((b, p_s), (b', p_d), h((p_s, p_d)) \circ l_a) \end{aligned}$$

The set $I_{BS}(b)$ represents the set of tuples of FQSN and port pairs selected by the *BusSelector* b, i.e. for a tuple (l, p) the signal with the FQSN l is emitted from b on port p. Each segment c whose FQSN either exactly matches or is contained in a segment with $FQSN(c) = l_{pre}$ is expanded over b, and their label is updated accordingly. This way we assure that signal segments contained within a selected bus signal are also propagated over the BusSelector block.

BusAssignment blocks are able to manipulate existing bus signals by exchanging a signal currently present in a bus with another signal with identical data type.

Definition 5.14 (BusAssignment Block Expansion Function).
Let $\mathcal{A}(b)$ be the set of tuples of fully qualified signal names of the signals and their incoming ports assigned by BusAssignment b and p_{bus} the port of the incoming bus signal. The expansion function for BusAssignment blocks (BA)

$$expand_{BA}(c = (L_s, L_d = (b, p), l))$$

is defined as

$$\begin{aligned} expand_{BA}(c) = \{c_a | \\ succ_c(c, c_a) \land \exists\, l' : (l', p_s) \in \mathcal{A}(b)\} \bigcup \{(c, c_c) | \\ succ_c(c, c_c) \land p_{bus} = p \land \forall\, (l', p') \in \mathcal{A}(b) : l' \neq l\} \\ with\ c_a = ((b, p_s), (b', p_d), h((p_s, p_d)) \circ l'|^2) \\ with\ c_c = ((b, p_s), (b', p_d), h((p_s, p_d)) \circ l|^2) \end{aligned}$$

The expansion function $expand_{BA}$ is split into two parts. The first part handles the expansion of segments that enter b through one of the ports $p \neq p_{bus}$. The signals of these segments replace an existing signal in the bus signal entering b via p_{bus}. A signal is replaced by creating a segment c_a and setting its FQSN to the FQSN of the signal it is assigned to, which is retrieved from the set $\mathcal{A}(b)$. The second part handles the termination of all signals that are assigned by b but were received via p_{bus} as well as the continuation of all segments c_c that represent signals not assigned by b.

Besides the BusCreator, BusSelector and BusAssignment block, there exist further bus-capable blocks. These blocks are handled by the following expansion function.

Definition 5.15 (Bus-capable Block Expansion Function).
The expansion function for bus-capable blocks (BCB)

$$expand_{BCB}(c)$$

is defined as

$$expand_{BCB}(c) = expand_{VB}(c)$$

Virtual and nonvirtual bus-capable blocks continue an incoming bus signal with the same signal characteristics. In terms of data dependence, we handle nonvirtual bus-capable blocks like virtual blocks, as the signals in the emitted copy are dependent on their respective original signals.

One further refinement of the expansion function for bus-capable blocks can be applied in the context of Switch and MultiPortSwitch blocks. These blocks choose one incoming signal to be propagated during a step of the model simulation based on the signal value received at a control port of the switch. Figure 5.6 shows a simple model that uses a Switch block for selective signal propagation. The Switch block has a logical condition attached to its control port (inport 2). If the value received via its control port exceeds the value 4, the signal received at the first port is propagated. Otherwise, the signal received

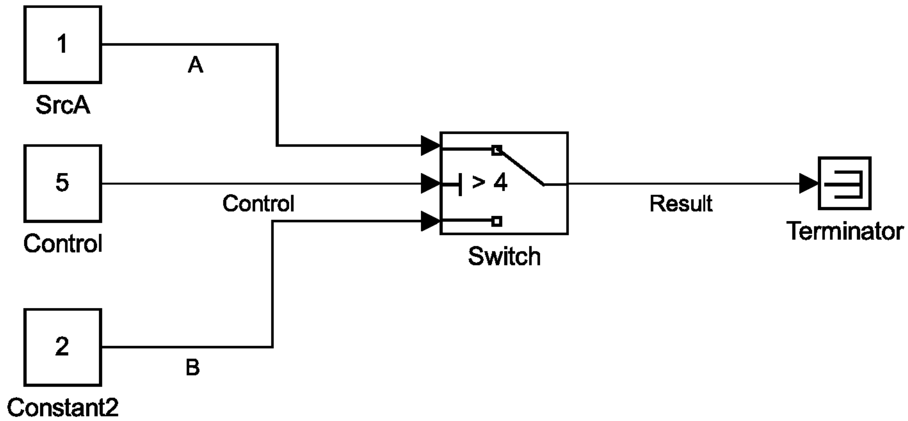


Figure 5.6: Switch block controlling signal propagation

at the third inport is propagated. As the evaluation of the control condition is run-time dependent, we need to trace both signals over the Switch block while terminating the received control signal. MultiPortSwitch blocks can have an arbitrary amount of incoming signal propagation candidates.

Definition 5.16 (Switch/MultiPortSwitch Block Expansion Function)**.**
Let $p_c \in P$ be the control port of a Switch or MultiPortSwitch block b. The expansion function for Switch/MultiPortSwitch blocks (SW)

$$expand_{SW}(c = (L_s, L_d = (b, p), l))$$

is defined as

$$expand_{SW}(c) = \begin{cases} \emptyset, & \textit{if } p = p_c \\ expand_{BCB}(c), & \textit{otherwise} \end{cases}$$

The expansion function $expand_{SW}$ terminates the signal received at the control port p_c of the respective switch block b. For Switch blocks, this is always the second inport of the block, while for MultiPortSwitch blocks the control port is always the first inport.

Signal graph construction To compute all signals and signal paths of a Simulink model $\mathcal{M}$, we initially create a signal segment for each signal that is emitted from each nonvirtual block, virtual block without inports and all BusCreator blocks represented by the set $B_{start} \subset B$. By iteratively applying the defined rules in a fix point depth-first search on the resulting signal segments, we can compute all related signal segments that describe the propagation of the signals emitted from the blocks $b \in B_{start}$. When selecting a rule to apply on a signal segment $c = (L_s, L_d, l)$, we always choose the most fitting rule with respect to the block described by the destination tuple L_d of c. We can then construct the signal paths of each signal starting in b, by computing all transitive arrangements of the computed related signal segments. If an expansion function returns multiple expanded signals segments $|C_c| > 1$ due to multiple lines leaving at a specific outport, the current signal path is copied $|C_c| - 1$ number of times, so that each segment in C_c can be added to a distinct path. By repeating this process for each $b \in B_{start}$, we can compute all signal paths S^P for a Simulink model $\mathcal{M}$ and associate them to their respective signals in S.

As mentioned in Section 2.2.1, signals can also be exchanged using a DataStoreMemory block via indirect signal flow but were not covered by an explicit expansion function. A DataStoreMemory block can save an arbitrary signal via a DataStoreWrite block. The currently stored value of the datastore can be retrieved using a DataStoreRead block. As the actual value that is read via a DataStoreRead block depends on the sorted order of the model, signals exchanged via a datastore need to respect this order.

Definition 5.17 (Signals Exchanged via DataStoreMemory Blocks).
Let $B_{DS}(b_{DS}) = B_{DR}(b_{DS}) \cup B_{DW}(b_{DS})$ be the set of DataStoreRead/Write blocks associated to a DataStoreMemory block b_{DS} with $B_{DR}(b_{DS})$ being the set of DataStoreRead blocks and $B_{DW}(b_{DS})$ being the set of DataStoreRead blocks associated to b_{DS}.

A signal s is exchanged between a DataStoreWrite block $b_{DW} \in B_{DW}(b_{DS})$ and a DataStoreRead block $b_{DR} \in B_{DR}(b_{DS})$ if one of the following conditions holds:

1. $b_{DW} \leq_{SO} b_{DR} \ \wedge \neg\exists\ b'_{DW} \in B_{DW}(b_{DS}) : b_{DW} \leq_{SO} b'_{DW} \leq_{SO} b_{DR}$
2. $\neg(b_{DW} \leq_{SO} b_{DR}) \wedge \neg\exists\ b'_{DW} \in B_{DW}(b_{DS}) : b'_{DW} \leq_{SO} b_{DR} \vee b_{DW} \leq_{SO} b'_{DW}$

A signal $s = (L_s, P_s)$ emitted from a DataStoreWrite block b_{DW} to a set of DataStoreRead blocks $B_{DR}(b_{DW})$ is defined as follows:

- $L_s = (b_{DW}, \perp)$
- $P_s = \bigcup_{b_{DR} \in B_{DR}(b_{DW})} p_{DS}(b_{DW}, b_{DR})$
- $p_{DS}(b_{DW}, b_{DR}) = (L_s, L_d = (b_{DR}, \perp), \langle (L_s, L_d, l_{DS}) \rangle)$

Here, l_{DS} represents the name of the datastore associated with b_{DW}.

The union of signals defined in Definition 5.17 for all DataStoreWrite blocks and the signals constructed via expansion functions is represented by the set S.

Using the set of signal paths S^P associated with the signals of S, we can formulate a flow sensitive definition for data dependence.

Definition 5.18 (Flow Sensitive Data Dependence).
Let S^P be the set of signal paths of a Simulink model $\mathcal{M}$. A block b_2 is data dependent on a block b_1 iff $\exists\ s^p = (L_s, L_d, C = \langle c_1 ... c_n \rangle) \in S^P$ such that $\exists\ i, j \in \mathbb{N}$ so that holds:

- $c_i = (L_{s_i} = (b_1, p_{out}), L_{d_i}, l_i)$
- $c_j = (L_{s_j}, L_{d_j} = (b_2, p_{in}), l_j)$
- $i <= j <= n$

Following this definition, a block b_2 is data dependent on another block b_1 if there exists a signal path s^p in the model, which contains a signal segment c_i starting at an outport of b_1 that either directly ends at b_2 or contains another segment c_j: $i < j$ ending at b_2.

5.3.3 Control Dependence in MATLAB/Simulink

MATLAB/Simulink also allows the modeling of conditionally executed model parts, resulting in conditional execution contexts during model simulation (see Section 2.2.2). The Switch block handled in Definition 5.16 is an example for a block introducing conditional execution contexts in a model (see Definition 2.1).

To describe control dependencies in a MATLAB/Simulink model, we need information about the conditional execution contexts of the model [116]. While Reicherdt et al. introduced an algorithm to calculate the execution contexts of a model [116]; we implemented a method as part of the MATLAB/Simulink tool adapter of the artshop framework (see Section 3.3.1) that automatically extracts all execution contexts and the sorted order of nonvirtual model elements from the simulation environment of MATLAB/Simulink. This information can be used to calculate control dependencies of a MATLAB/Simulink model in the same fashion as proposed by Reicherdt et al. [116].

5.4 Slicing Simulink Models

We will now introduce how we incorporated the flow-based definition for data dependence into a slicing algorithm. As the algorithm presented by Reicherdt et al. [116], our algorithm calculates a slice of a model on the dependence graph of the model via reachability analysis. The dependence graph is derived from introduced data and control dependence relations.

5.4.1 Building the Dependence Graph

While building the dependence graph G_D for a MATLAB/Simulink model $\mathcal{M}$ we need to assure that all signal paths of each individual signal $s \in S$ are encoded in G_D. The nodes of G_D are created by iterating over all signal paths S^P of $\mathcal{M}$ and adding a node v for each block encountered on a signal path s^p and an edge for signal segment connecting two blocks on s^p. After that, we merge all nodes in G_D that represent a block in the set of nonvirtual, non-bus-capable block $B_{NVB} \in B$ from $\mathcal{M}$ and update the edges ending at these nodes accordingly. Finally, we add edges between control dependent nodes in G_D.

Definition 5.19 (MATLAB/Simulink Dependence Graph).
A dependence graph $G_D = (V_D, E_D, f_D)$ for a MATLAB/Simulink model $\mathcal{M}$ is a directed graph with

- *V_D being the set of nodes of the graph representing blocks in $\mathcal{M}$*

- *E_D being the set of tuples $V_D \times V_D$ representing directed dependence edges between the nodes in V_D*
- *$f_V : V_D \rightarrow B$ being a function mapping the nodes of V_D to the blocks in $\mathcal{M}$*
- *$f_E : E_D \rightarrow C \cup \{\perp\}$ being a function mapping the edges in E_D to the segments in C or to $\perp$ if an edge represents a control dependence relation*

The encoding of all signal paths in G_D leads to multiple nodes in V_D being mapped to the same block. We further define the mapping function f_V that maps nodes from V_D to the blocks in $\mathcal{M}$ and f_E mapping edges in E_D to the signal segments.

Encoding all signal paths in G_D and only merging nonvirtual, non-bus-capable blocks enables the traceability of atomic signals across bus-capable blocks as described in the context of Definition 5.15.

5.4.2 Slice Computation

Prior to slice computation, a slicing criterion must be chosen as a start point of the reachability analysis in the dependence graph G_D of a model $\mathcal{M}$. The slicing criterion is a set of blocks $B_{SC} \in B$ with $B_{SC} = \emptyset$ resulting in an empty slice. Otherwise, the slicing algorithm returns a connected slice of the model, which contains blocks that are directly or transitively data/control dependent on the blocks of the slicing criterion.

Definition 5.20 (MATLAB/Simulink Model Slice)**.**
A slice $\mathcal{M}_S(B_{SC}) = (B_S, C_S)$ of a Simulink Model $\mathcal{M} = (B, P, L, I, f_p, h)$ for a slicing criterion B_{SC} contains a set of blocks $B_S \subseteq B$ and a set of signal segments C_S for that holds:

- $\forall$ *$b \in B_S$: b is directly or transitively data/control dependent on B_{SC} (forward) or vice versa (backward)*
- $\forall$ *$c = (L_s = (b_s, p_s), L_d = (b_d, p_d), l) \in C_S : b_s, b_d \in B_S$*

Besides the reachable blocks, the slice also contains all signal segments that were passed to reach the blocks in B_S. This preserves the signals that were relevant during the computation of the slice and can be used to calculate all other elements contained in the slice, e.g. lines and ports, for latter visualizations or further use during other analyses. A slice $\mathcal{M}_S(B_{SC}) = (B_S, C_S)$ can be computed by performing a forward or backward reachability analysis on the nodes of the dependence graph from the nodes corresponding to the blocks of the slicing criterion.

Algorithm 1 shows a sketch of the slicing algorithm in pseudo code. Besides the slicing criterion B_S, the algorithm receives the dependence graph G_D and a boolean value d_F indicating if the slice has to be performed in forward ($d_f = 1$) or backward ($d_f = 0$) direction. The algorithm starts by mapping the blocks of the slicing criterion to the nodes of the dependence graph and storing them within a worklist (line 1). After the initialization of the sets of reached nodes and edges in line 2 - 3, the algorithm continues

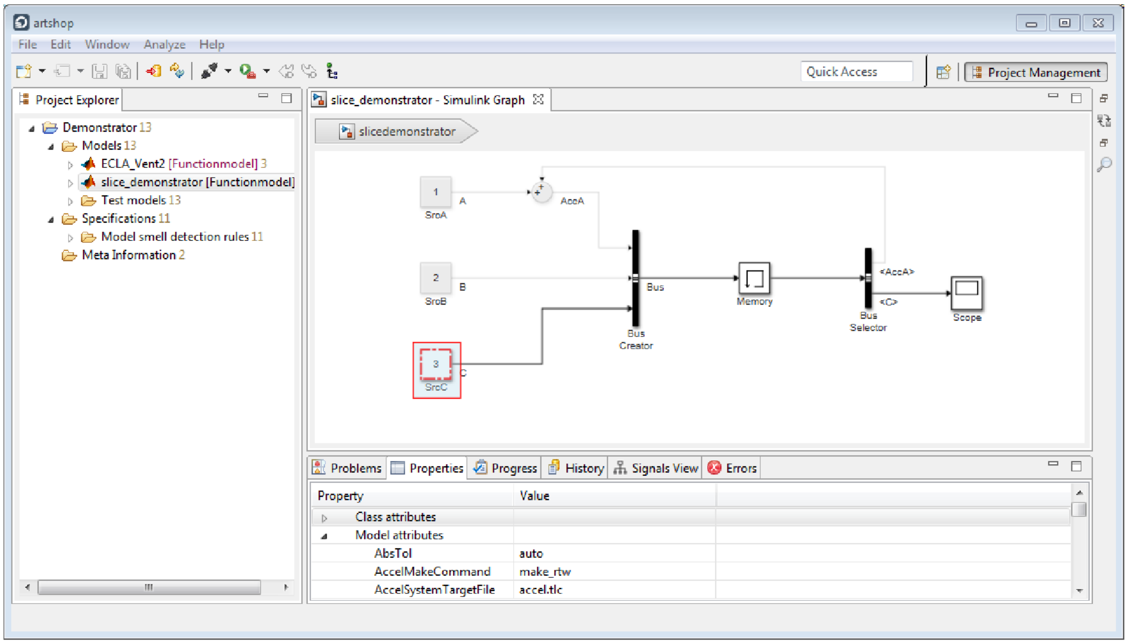


Figure 5.7: Visualization of the forward slice on the block SrcC in artshop

by processing the nodes in the work list (line 5 - 24). If a forward slice is computed ($d_f = 1$), the algorithm determines all outgoing edges of a node v and adds all nodes reachable from these edges to the set $reached_v$, while also storing each used edge to determine a reachable node in the set $reached_e$ (line 6 - 14). After all nodes have been processed, the worklist is set to the nodes that have been added during the reachability analysis of the nodes currently contained in the worklist (line 25). This procedure is continued until the worklist no longer contains elements. The resulting slice $\mathcal{M}_S(B_{SC})$ is then computed by mapping the nodes and edges contained in $reached_v$ and $reached_e$ to their respective blocks and signal segments. If a backward slice shall be computed ($d_f = 0$), the algorithm works analogous but traverses the edges of G_D in backward direction (line 15 - 23).

5.4.3 Presentation

We have integrated the slicing algorithm as an extension component into the artshop framework. A slice can directly be computed from a selection of blocks in the graphical view of a MATLAB/Simulink model provided by the respective tool adapter (see Section 3.3.1).

From this view, a slicing criterion can be chosen via a selection of blocks and slice calculation can be triggered. Slicing is performed on the model representation imported by the tool adapter. The result is displayed by either hiding elements that are not contained in the slice or fading them into the background, as it can be seen in Figure 5.7. A calculated slice can be exported into a stand-alone MATLAB/Simulink model. During

Algorithm 1: Slicing algorithm

Input: Slicing criterion B_{SC}, $G_D = (V_D, E_D, f_D)$, Slice direction forward $d_f \in \mathbb{B}$
Output: Slice $\mathcal{M}_S(B_{SC})$

```
worklist = { v | v \in V_D \wedge f_V(v) \in B_{SC} };
reached_v = \emptyset;
reached_e = \emptyset;
while worklist \neq \emptyset do
    foreach v \in workList do
        if d_f then
            out_v = {(v, v') | (v, v') \in E_D};
            foreach e = (v, v') \in out_v do
                if v' \notin reached_v then
                    reached_v = reached_v \cup {v'};
                    reached_e = reached_e \cup {e};
                end
            end
        end
        else
            in_v = {(v', v) | (v', v) \in E_D};
            foreach e = (v', v) \in in_v do
                if v' \notin reached_v then
                    reached_v = reached_v \cup {v'};
                    reached_e = reached_e \cup {e};
                end
            end
        end
    end
    worklist = reached_v \setminus worklist;
end
B_S = { b | \exists v \in reached_v : f_V(v) = b };
C_S = { c | \exists e \in reached_e : f_E(e) = c };
return \mathcal{M}_S(B_{SC}) = (B_S, C_S)
```

this procedure, a copy of the initial model is created and subsequently all lines and blocks that are not contained in the slice are deleted by issuing commands via the interface provided by the tool adapter. By deleting lines, block configurations of BusCreator and BusSelector blocks may become inconsistent as previously available signals are missing and prevent the model from being compiled. We have implemented a cleanup script that fixes inconsistent block configurations by using the signal information present in the respective slice. The resulting model should be in an executable state if a backward slice

Table 5.2: Average forward slice sizes of the flow-based slicing algorithm

Model	t_{G_D} (ms)	DD			DD + CD		
		$\bar{t}_{AVG}$ (ms)	$\mathcal{M}_{AVG}$ (%)	σ (%)	$\bar{t}_{AVG}$ (ms)	$\mathcal{M}_{AVG}(m)$ (%)	σ (%)
MAV	13.33	0.72	19.46	17.83	1.32	24.56	22.89
DAS	14.85	0.41	8.98	12.72	1.40	42.77	25.0
EL	15.1	0.15	2.80	4.48	0.53	10.51	12.47
PI	79.6	0.89	3.51	8.66	1.55	5.33	12.54
ECLA	200.67	31.7	42.06	26.41	34.64	44.08	26.01

is exported, while in forward slices blocks might be missing that do not depend on the slicing criterion but are needed to execute the model.

5.5 Evaluation

In the following, we present the evaluation results for the flow-based slicing algorithm from the previous section. The evaluation has been performed on the same system as the evaluation presented in Section 3.5 and with the models introduced in Section 3.5.1.

We already compared the evaluation results of the models analyzed by Reicherdt et al. in [116] with the results of the flow-based slicing approach in [59]. As no bus signals were used in these models, no differences between the average slice sizes of the line-based and flow-based slicing algorithms could be detected. In this work, we used the same metric as Reicherdt et al. to compare the precision of both approaches, the average slice size.

Definition 5.21 (Average Slice Size of a MATLAB/Simulink Model)**.**
Let $\mathcal{M} = (B, P, L, I, f_p, h)$ be a MATLAB/Simulink model. The average slice size $\mathcal{M}_{AVG}(B)$ on the blocks of $\mathcal{M}$ is then defined as:

$$\mathcal{M}_{AVG}(B) = \frac{1}{|B|} \sum_{b \in B} \frac{|\mathcal{M}_S(\{b\})|}{|B|}$$

Following this definition, $\mathcal{M}_{AVG}(B)$ represents the average percentage of blocks contained in the slices of all blocks in $\mathcal{M}$.

We first assess the performance of our approach by analyzing key performance indicators (KPI), e.g. dependence graph computation time, average slice time and slice sizes on a set of industrial size models that extensively use bus signals. In comparison to the evaluation results presented by Reicherdt [115, 116], the models used during this evaluation are 3-27 times bigger and extensively use bus signals. Second, we show the impact of the flow-based slicing approach, by comparing the average slice sizes between the flow-based and a line-based data dependence relation as initially introduced by Reicherdt et al. [116]. We could not evaluate the algorithm presented by Pantelic et al. [108], as the MATLAB script realizing their approach did not run successfully on our set of evaluation models.

Table 5.3: Average backward slice sizes of the flow-based slicing algorithm

Model	t_{G_D} (ms)	DD $\bar{t}_{AVG}$ (ms)	DD $\mathcal{M}_{AVG}$ (%)	DD σ (%)	DD + CD $\bar{t}_{AVG}$ (ms)	DD + CD $\mathcal{M}_{AVG}(m)$ (%)	DD + CD σ (%)
MAV	14.85	0.85	19.71	21.97	2.41	23.99	23.14
DAS	15.1	0.58	14.35	18.37	0.77	42.26	36.94
EL	13.33	1.09	8.0	11.9	1.30	9.88	12.38
PI	79.6	2.24	6.08	7.03	2.77	6.42	6.96
ECLA	200.67	34.5	48.63	38.26	37.69	51.10	38.17

5.5.1 Evaluation of the Flow-Based Slicing Algorithm

Table 5.2 and 5.3 show the empirical evaluation results of the slicing algorithm on the evaluated models. For each model, the table displays the time t_{G_D} needed to construct the dependence graph G_D of the model, as well as the average slice size $\mathcal{M}_{AVG}$ and its corresponding standard deviation σ. These KPIs are displayed for all slices in forward and backward direction, based on data dependence (DD) and data + control dependence (DD+CD).

For all models, the dependence graph can be constructed in under 300 ms and the average time $\bar{t}_{AVG}$ to compute a slice in either forward or backward direction does not exceed 40 ms. It can be noted that while the PI and ECLA model have similar overall sizes (8224 blocks vs. 8639 blocks), the average construction time of the dependence graph as well as the average computation of a slice differ by quite a lot. As the performance of the slicing algorithm is heavily influenced by the amount of actual signals (not lines) contained in a model, we analyzed the complexity of the signals in the evaluation models. The results of this analysis are shown in Table 5.4. When comparing the amount of signals, bus signals and signal segments in both models, it becomes apparent that the ECLA model contains about 1000 additional actual signals that are routed through 4-times more bus signals in comparison to the the PI model. The amount of signal segments contained in bus signals also highlights this fact, as 72 % of all segments are encapsulated

Table 5.4: Bus signal complexity of the evaluated models

Model	Signals	Bus signals	Signal seg. (cont. in bus)	Avg/Min/Max cont. signals	Avg/Min/Max bus depth
DAS	413	24	2397 (606)	5/2/16	1/1/1
EL	714	27	1302 (327)	9/3/25	1/1/1
MAV	553	15	2584 (1482)	11/4/48	1/1/2
PI	4370	48	10989 (1820)	8/3/71	1/1/3
ECLA	5186	211	36393 (26440)	15/2/348	1/1/6

Table 5.5: Comparison of forward and backward slices of a line-based and the flow-based slicing approaches

Model	Forward				Backward			
	DD		DD + CD		DD		DD + CD	
	$\mathcal{M}^L_{AVG}$ (%)	$\mathcal{M}^F_{AVG}$ (%)	$\mathcal{M}^L_{AVG}$ (%)	$\mathcal{M}^F_{AVG}$ (%)	$\mathcal{M}^L_{AVG}$ (%)	$\mathcal{M}^F_{AVG}$ (%)	$\mathcal{M}^L_{AVG}$ (%)	$\mathcal{M}^F_{AVG}$ (%)
MAV	33.11	19.46	40.65	24.56	33.09	19.71	39.97	23.99
DAS	31.11	8.98	59.98	42.77	31.13	14.35	59.01	42.26
EL	25.77	2.80	31.15	10.51	24.58	8.00	28.61	9.88
PI	9.91	3.51	10.48	5.33	9.90	6.08	10.29	6.42
ECLA	60.92	42.06	64.72	44.08	60.92	48.63	64.72	51.10

in a bus, with the biggest bus signal containing 348 individual signals in up to 6 nested bus signals.

When comparing the average slice size across both directions, it can be noted that the average slice size is heavily influenced by the actual composition of the respective model. For example, the average slice sizes of the closed loop models (DAS, ECLA) is much higher than for the other models. Due to feedback signals, strongly connected components are formed in the dependence graph, resulting in an overall increase of slice sizes. Slice sizes are further influenced by control dependence relations. While the average slice size in forward direction of the DAS model is 8.8 %, it increases to 42.77 % if control dependence relations are considered during slice computation. The increase of the average slice size from DD+CD depends on the amount of blocks influencing the control flow of the respective models. Although the average slice sizes for each model in forward and backward directions do not diverge much from each other, the standard deviation of the average slice size indicates that the distribution of slice sizes between forward and backward direction differ by up to 10-12 %. This is consistent with the findings of Reicherdt [115].

Across all evaluated models, the flow-based slicing algorithm achieves an average slice size of 25.45 % in forward and 26.73 % in backward direction when both data and control dependencies are considered by the algorithm. These findings are consistent to the evaluation results of Reicherdt [115] that were obtained from the application of his slicing techniques to MATLAB/Simulink models containing bus signals.

5.5.2 Impact of Flow Sensitive Slicing

To show the impact of the flow-based data dependence relation, we implemented the line-based data dependence relation introduced by Reicherdt et al. [116]. In the following, we compare the average slice sizes of both approaches against each other.

Table 5.5 displays the average slice sizes of both approaches in forward/backward direction in the same fashion as in the previous section. When comparing the average slice sizes in forward direction, whereby only considering data dependencies (DD), the average improvement of the flow-based against the line-based one is around 16.80 % in favor of the flow-based approach. Adding control dependencies to these slices degrades the improvement slightly to approx. 15.95 %. Depending on the size of the model, this results in an absolute average block difference of 165 blocks for the MAV model to 1783 blocks in case of the ECLA model (DD+CD). The maximal absolute deviation recorded during the evaluation of the approaches in forward direction, is one slice of the ECLA model, covering 71.81 % (6208 blocks) of the model using the line-based approach. A slice computed with the flow-based approach with the same slicing criterion only resulted in a slice coverage of 0.13 % (12 blocks). As the flow-based slicing approach considers signals terminating at BusSelector, BusAssignment and BusCreator blocks, this slice represents one of the cases where a flow-based approach vastly outperforms the line-based approach.

Comparing both approaches in backward direction, similar results can be observed as for the slices in forward direction. Average differences across both approaches are 12.57 % (DD) and 13.79 % (DD+CD) respectively, slightly lower than the average differences for slices in forward direction. Therefore, the average absolute difference also decreases slightly to 164 in case of the DAS model and up to 1177 blocks in case of the ECLA model (DD+CD). The maximal recorded absolute deviation in backward direction was again encountered during the analysis of the ECLA model. A slice was found that contained 90.02 % (7793 blocks) of all blocks of the model for the line-based approach and 6.94E-4 % (6 blocks). Again, this slice was related to a signal being exchanged at a BusAssignment block.

Overall, the flow-based slicing approach is superior to the line-based data dependence approach, if the analyzed models contain bus signals. As we have already noted in [59], for models that do not contain bus signals both approaches result in the same average slice sizes.

5.6 Extension for MATLAB/Stateflow

A block type not specifically handled by the signal reconstruction and slicing algorithms are Chart blocks introduced by the MATLAB/Stateflow toolbox (see Section 2.2.3). While a Chart block is currently treated as a nonvirtual block, i.e. all signals entering it may potentially influence all outgoing signals, the interface of these blocks typically grows linearly with increased complexity of the modeled behavior. As for Subsystem blocks, the dependencies between the in- and outports of a Chart block depend on its internal composition. To increase the precision of the Simulink slicing algorithm, we present a proof-of-concept algorithm to determine the dependencies between in- and outports of a MATLAB/Stateflow Chart block by analyzing the statechart contained within a Chart block.

To trace dependencies between the in- and outports of a Chart block, the usage of the in- and output variables within the statechart have to be analyzed, similar to traditional dependency analysis. As states and transitions may contain operations that manipulate the variables of the statechart, we can compute the transitive relationships of variables by analyzing assignments made to these variables.

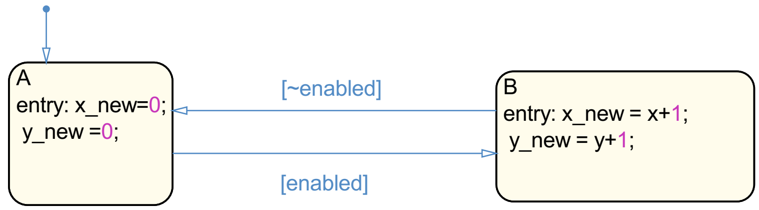


Figure 5.8: Simple MATLAB/Stateflow state chart

Consider the statechart shown in Figure 5.8. The statechart has three inports named x, y and *enabled* as well as two outports named x_new and y_new. These ports are mapped to equally named in- and output variables used within the statechart. To determine which input variables are used to calculate the value of output variable x_new, we first determine all assignments made to this variable and analyze the variables referenced within these assignments.

Definition 5.22 (Stateflow Assignment)**.**
An assignment to a variable v within a MATLAB/Stateflow statechart is expressed as a tuple (v, R) with R being the set of variables referenced on the right-hand side of the assignment to compute the new value of v.

The assignment

$$x\_new = x + 1;$$

contained in state B can be expressed as the assignment $a = (x\_new, \{x\})$. Therefore, we can deduce that x_new depends on variable x. As this assignment has been made in a state that is not part of the initial states of the state chart, we need to determine all paths from the initial state A to state B. Furthermore, we need to treat all variables referenced within the guards of transitions contained within these paths as an influence to x_new as they affect the reachability of state B, yielding a type of control dependence for statecharts similar to the one discussed in Section 5.3.3 for blocks in MATLAB/Simulink. In the example, the single path from A to B uses a transition referencing the variable *enabled* within its guard. Therefore, this variable is also added to the set of variables that x_new depends on. Thus, the output variable x_new within the example statechart would depend on the set of variables $\{x, enabled\}$.

Typically, statecharts also contain transitive relationships among variables that need to be considered during dependency detection. We have implemented a proof-of-concept algorithm that can transitively calculate an over-approximation of the dependencies for all variables used within a given statechart.

Algorithm 2 shows the variable dependence algorithm in pseudo code. It returns a map D that maps each variable $v \in V$ to a set of variables that it potentially depends on. The algorithm first collects all assignments for each variable $v \in V$ by using the *assignments* function on each variable and storing it in the set of all assignments A as well as initializing the dependence map D with an empty set for each variable (line 2-5).

Algorithm 2: Statechart variable dependence detection

Input: Set of variables V used in a statechart
Output: Dependence map D for each variable $v \in V$

```
A = ∅;
foreach variable v ∈ V do
    D(v) ← ∅;
    A = A ∪ assignments(v);
end
foreach a = (v, R) ∈ A do
    ref_v = R ∪ {v};
    P = paths(a);
    if P ≠ ∅ then
        P = P ∪ cycles(a);
    end
    foreach path p ∈ P do
        foreach state s on p do
            foreach a′ = (v′, R′) ∈ A on s do
                if v′ ∈ ref_v then
                    ref_v = ref_v ∪ R′;
                end
            end
            foreach transition t leading to s in p do
                foreach a′ = (v′, R′) ∈ A on t do
                    if v′ ∈ ref_v then
                        ref_v = ref_v ∪ R′;
                    end
                end
                foreach r ∈ guard(t) do
                    ref_v = ref_v ∪ {r};
                end
            end
        end
    end
    D(v) ← D(v) ∪ ref_v;
end
return D
```

Following, the algorithm traverses over each assignment $a = (v, R) \in A$ and collects all variables affecting v in the set ref_v. The variables of ref_v are calculated by first computing all paths P potentially leading to an execution of the assignment a from one of the initial states, using the *paths* function. Paths are calculated on a graph of the flattened statechart. If there exists at least one path leading to the potential execution of the current assignment a, we also consider all cycles leading to an execution of a during dependence analysis to incorporate all possible dependencies to variables residing on these cycles. Cycles are retrieved with the *cycle* function using an implementation[1] of the cycle detection algorithm proposed by Szwarcfiter et al. [150] (line 9-11).

The set ref_v is further populated by iterating over all paths $p \in P$ and adding all references R' in assignments of the form $a' = (v', R'), v' \in ref_v$ encountered in transitions and states on the path p in descending order with respect to the path and the order of assignments on the respective states/transitions (line 14-24). Additionally, all variables referenced within the guards of transitions p are also added to ref_v using the *guard* function (line 25-27). By considering all references to v stored in the set ref_v, we compute the transitive closure of all references and assignments encountered on a given path p.

In its current form, the algorithm does not support the analysis of user-defined functions but still recognizes the parameters of a function whose result is assigned to a variable v as dependencies for v. Furthermore, the algorithm currently cannot correctly evaluate state charts containing parallel states because changes to a variable in one state cannot be related to the other state as part of a path.

The algorithm has been implemented within the artshop framework and integrated into the slicing algorithm. The model representation of the MATLAB/Simulink tool adapter (see Section 3.3.1) is used to access MATLAB/Stateflow elements. To extract information about assignments and references of states and transitions, we first implemented a parser capable of extracting relevant code sections from the labels attached to these elements. We use a built-in MATLAB function to create an abstract syntax tree (AST) for extracted code sections, e.g. guards and actions for transition labels and state operations for state labels. The AST is then used to extract variable assignments and references from the respective code sections. We further provide a wrapper that maps a given inport/outport to the respective input/output variable, performs the dependency analysis shown in Algorithm 2 and maps the variables of the result set to their respective inport/outport.

We evaluated the impact of the shown algorithm by calculating the inport-outport relationships for 10 Stateflow statecharts taken from industrial and academic models. In total, the analyzed Stateflow models contain 39 outports and 61 inports. For each outport we calculated all inports that a given port depended on and vice versa for each inport. The data dependencies obtained during slicing can be reduced when the algorithm determines that an outport depends only on a subset of the inports and vice versa. This was the case for 5 of the 10 analyzed statecharts. In forward direction, the algorithm determined that 21 of the 61 inports that do not influence all outports of their respective Stateflow statecharts leading to an average weighted improvement of the data dependency relation between the in- and outports of the analyzed Stateflow statecharts of 17.47 %.

[1] `http://jgrapht.org/`

In backward direction, 24 of the 39 outports could be identified that do not depend on all inports of their respective statecharts. This leads to an average weighted improvement of the data dependency relation between outports and inports of around 20.38 %.

This empirical evaluation shows that the technique can further improve the data dependency relation if MATLAB/Stateflow Chart blocks are used within a model $\mathcal{M}$. However, the effectiveness is highly dependent on the inner structure of the analyzed statechart, as the desired behavior might require all inports to influence all outports of the respective statechart.

5.7 Conclusion and Future Work

In this chapter, we extensively discussed signal flow within MATLAB/Simulink models and proposed an algorithm to reconstruct all atomic and bus signals exchanged within a MATLAB/Simulink model. The signals derived by this algorithm were used to define a signal flow-based data dependence relation for the use within a slicing algorithm. Using the data dependence relation in combination with control dependence information extracted from MATLAB/Simulink, we can construct the dependence graph of a MATLAB/Simulink model. A slice on the model can then be computed by performing a reachability analysis on this graph in either forward or backward direction. Slices computed in the artshop framework are directly visualized within the graphical viewer provided by the MATLAB/Simulink tool adapter and can again be exported as a stand-alone MATLAB/Simulink model.

The evaluation of the flow-based slicing algorithm showed that, on average, the proposed slicing algorithm can reduce the size of a MATLAB/Simulink model by about 75 %. This makes the slicing algorithm particularly useful during debugging, testing or change-impact analysis of MATLAB/Simulink models. Further, we showed that the use of a flow-based data dependence relation is superior to a line-based relation on models containing bus signals. On average, the difference between the average slice sizes using a flow-based or line-based data dependence relation is about 12-15 % of all model elements. As bus signals are commonly used in large scale academic and industrial size models, the use of our proposed flow-based data dependence relation is a reasonable way of reducing slices of MATLAB/Simulink models.

The slicing algorithm can further be improved by also considering the initial structure of Chart blocks during the slicing algorithm to derive a dependence mapping between the in- and outports of a Chart block. We proposed a proof-of-concept algorithm that derives such a mapping based on the usage of Stateflow variables within the statechart contained in the Chart block. An evaluation of the prototypical algorithm showed that the computed mapping on average provides an improvement of approx. 20.38 % in comparison to the treatment of a Chart block as a nonvirtual block, where all input signals are assumed to influence all output signals.

By extending the definition of the slicing criterion to signals emitted by the blocks selected as part of the slicing criterion, the dependencies of specific signals emitted by a block could be explored. This would resemble the definition of the slicing criterion

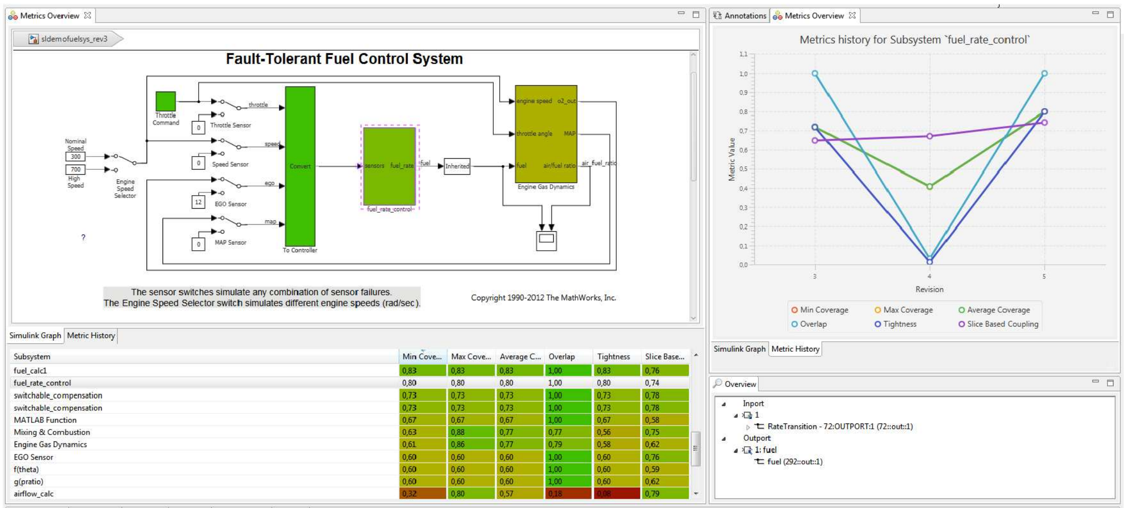


Figure 5.9: Result view of the complexity metric analysis on the example model *sldemo_fuelsys*

initially proposed by Weiser [160], who defined the slicing criterion as a statement and a set of variables that occur within this statement.

5.7.1 Further Applications

The slicing algorithm and the reconstructed signal information can further be used to construct analyses that need information about the signal characteristics and dependence relations within a model. As already noted during the discussion of related work, the static value range analysis developed by Dernehl et al. uses the signal information calculated by artshop to reconstruct the initial signal structure of the model [40, 41, 42, 43, 44].

Slice-based complexity metrics We used the slicing algorithm to implement slice-based cohesion and coupling metrics for Subsystem blocks. The metrics were derived from corresponding slice-based metrics on program code and adapted to the syntax of MATLAB/Simulink models. The cohesion metrics were derived from the initial definitions by Ott and Thuss [106], while the coupling metric was derived from a metric proposed by Harmann et al. [67]. In contrast to existing cohesion and coupling metrics for MATLAB/Simulink models [35, 36, 104, 105] that only define these metrics on a structural and not on a functional level, slice-based metrics include the semantic dependencies contained within the individual slices of the model. Further information about the benefits of slice-based metrics can be found in the empirical study on slice-based coupling and cohesion metrics published by Meyers and Binkley [99].

Figure 5.9 shows the result of the metric calculation on a slightly modified version of the example model *sldemo_fuelsys* included in MATLAB/Simulink. Results can be browsed in a sorted list, based on metric value while the trend of all metrics calculated for a selected Subsystem block can be seen over its evolution within the graph on the upper right-hand side. This is the first analysis that uses the history information of the repository to perform trend analysis. To show the trend analysis, we modified the initial version of the *Controller* subsystem in three versions to show the resulting change in metric values. Additionally, a view of the analyzed model is provided where each subsystem is colored corresponding to its calculated metric value. In future work, the adapted metrics from Ott and Thus as well as Harman et al. should be compared in a correlation analysis against existing cohesion and coupling metrics proposed by Dajsuren and Olszewska et al. [35, 36, 104, 105]. In addition, structural metrics of MATLAB/Simulink models, e.g. amount of blocks contained in a subsystem, should also be considered during this correlation analysis.

Model transformation The reconstructed signal information were further used to extract the signal dependencies between a selection of Subsystem blocks within a MATLAB/Simulink model and transform the selection into a corresponding SysML architectural model. As MATLAB/Simulink was one of the earliest adoptions of model-based development within the automotive domain, there usually do not exist further architectural description models of legacy MATLAB/Simulink models. With architectural modeling with SysML slowly gaining traction within the automotive domain, the implemented transformation process is a useful tool to derive an initial basis for the reconstruction of the functional architecture of a legacy MATLAB/Simulink in SysML. As no concept similar to bus signals exists within SysML, we flatten each bus and assign the fully qualified signal name to each signal created in the SysML model. The SysML model is created via a small tool adapter written for the modeling tool MagicDraw using its exposed external API.

Model smells Finally, the model smell detector, which is described in Chapter 7, also uses the reconstructed signal information to detect design flaws within the signal architecture of a MATLAB/Simulink model.

5.7.2 Future Work

In future work, the proof-of-concept dependence mapping for Stateflow charts needs to be extended to support the use of parallel states as well as the resolution of Stateflow functions. These extensions would enable the use of the algorithm on large scale Stateflow charts as these typically use the aforementioned concepts.

Another block currently treated as a nonvirtual block are Embedded-Matlab-Function blocks. These blocks contain a MATLAB script realizing the behavior of the block. By analyzing the dependence relations of the contained script, it would be possible to refine the dependence mapping of in- and outports of these block in the same way as for Chart blocks from MATLAB/Stateflow. The script information is already available within the representation created by the artshop tool adapter and used as part of a value range analysis of MATLAB code by Dernehl et al. [42]. Extending the slicing algorithm to this code could be realized by the use of standard slicing techniques for program code.

In combination with the static value range analysis developed by Dernehl et al., it would also be possible to extend the slicing algorithm to support dynamic slicing of MATLAB/Simulink models. This would further extend the usefulness of the slicing algorithm for debug and test purposes of MATLAB/Simulink models. A dynamic slicing approach could potentially be used to improve the performance of the static value range analysis by excluding the evaluation of irrelevant parts of the model, similar to the conditional execution behavior of MATLAB/Simulink.

While we already provide a special view of the sliced model, the slicing algorithm could be further used to create context sensitive views based on traceability links or variability information provided by other artifacts. For example, a view could be created that excludes all features unselected within a given variant description model created in pure::variants. In combination with a feature mapping to the blocks of a model, dependent blocks can be determined and excluded from a view to show only those parts of the model that are active based on the selected features within the variant description model. Further views can be created to improve change-impact analyses on MATLAB/Simulink models based on the history information available within the model repository of the framework.

6 Detection and Refactoring of Clones in MATLAB/Simulink Models

We already highlighted the complex nature of industrial size MATLAB/Simulink models, in Chapter 5. Due to the size and complexity of these models and the lifecycles they are developed in, quality defects like code clones that are commonly found in traditional software artifacts, e.g. source code, are also relevant for graphical dataflow diagrams such as MATLAB/Simulink models.

As in traditional software development, clones might be created in a multitude of ways during the application of model-based development. Copying model fragments within the graphical modeling environment provided by MATLAB/Simulink without creating suitable abstractions for controlled reuse of an already existing functionality, is a tempting way to quickly transfer functionality from one model location to another. Furthermore, using patterns or idioms for the solution of commonly encountered problems might also lead to clones within a model when applied in a wrong or naive way. Only rarely are clones created unintentionally by remodeling already existing functionality.

Cloned model fragments increase the effort needed to perform model maintenance, as changes to cloned model fragments need to be applied to all (potentially unknown) instances of the respective clone [38]. If a change is not propagated to all cloned model fragments, erroneous behavior might be introduced as highlighted by Juergens et al. in the context of cloned code [75]. Correct handling of model clones is even more important in the context of model-based development in the automotive domain, as models are heavily reused during the often product line oriented development processes [38]. Besides the relevance of clone detection for model maintainability, controlled reuse of model fragments and their traceability throughout a product line is at least equally important to increase the overall quality of model-based artifacts of a product line.

While techniques for model clone detection have received a fair amount of attention within the literature, most techniques focus on a specific type of clones, with the type of a clone specifying the degree of similarity between two sets of model elements. Typically, clone detectors use heuristics to identify clones to increase the overall performance, as solving the maximum common subgraph problem is NP-complete and therefore no polynomial algorithm exists for the computation of an exact solution [39]. Therefore, to analyze a model with regard to multiple type of clones, different tools have to be used that potentially produce disjoint or overlapping results in different output formats. In addition, detected clones still need to be resolved in a manual process. In this chapter, we present an approach to consolidate the results of multiple clone detectors and perform automatic clone resolution for a fixed set of clone types.

6.1 Overview and Outline

For the past years, clone detection in model-based software development, in particular for MATLAB/Simulink models, has been an active research area. Different techniques exist that detect model clones based on the similarity between a duplicate base and its duplicate occurrences. A taxonomy of clone types for data-flow models was first introduced by Gold et al. [64] that distinguishes between four clone types:

- **DF0**: Exactly copied model fragment
- **DF1**: Duplicate occurrence only differs with regard to layout or non-semantic properties
- **DF2**: **DF1** + duplicate occurrence additionally differs by literal values configured within blocks, e.g. the gain of a Gain block
- **DF3**: **DF2** + may include modifications to the duplicate base, e.g. added/deleted blocks/lines within a duplicate occurrence

Additionally, Bellon et al. [8] mention semantic clones as a fifth type in the context of code clone detection. While the similarity relation of the clone types introduced by Gold et al. is based on the structural similarity of a model, semantic clones can only be identified based on their realizing behavior.

There exist many approaches targeting specific clone types within MATLAB/Simulink models ranging from DF0 to DF3 [3, 38, 109] as well as approximate approaches based on model metrics [45] with diverse result sets. Result sets of certain techniques might contain clone groups that contain all clone occurrences with respect to a given clone base or only return an unordered set of clone pairs. We present an approach for sorting and merging clones detected by various duplicate detection techniques into an unambiguous result set, which reduces the overall amount of clones that need to be inspected. Within this approach, we combine two well-known detectors from the literature, namely ConQAT [38, 39] and gApprox [26], in addition to a novel copy-clone detector that is based on relative layout information. The result set of this combined detector consists of hierarchical clone groups that are consolidated across all applied clone detectors. As already noted by Gold et al., layout information provide an important factor during the clone detection in graphical data flow languages that is disregarded by both ConQAT and gApprox [64]. With the novel layout-based copy clone detector, we specifically target model fragments that were duplicated via the copy & paste actions available within the MATLAB/Simulink IDE.

To ease the process of clone resolution into reusable model fragments, we propose a process to generalize clones of type DF0-DF2 and transform them into a generic library block. Clone occurrences can then automatically be replaced by an instance of the created generic library block that is configured to match the initial behavior of the clone occurrence. The aforementioned clone detection process is specifically adapted for the detection of clones that can be refactored in a meaningful way, e.g. clones are detected on subsystem level and do not cross model hierarchy levels.

In the following, we will first discuss related work in the area of clone detection for MATLAB/Simulink models and then introduce the contributions of this chapter. After the introduction of our clone detection architecture at the beginning of Section 6.2, we will present the layout-based copy clone detector in Section 6.2.1 before introducing the clone consolidation procedure in Section 6.2.2. After we introduce the clone resolution procedure in Section 6.3 we will provide an evaluation of our approach in Section 6.4. Finally, we give an outlook on how the repository of the artshop framework can be utilized to perform cross-clone detection based on clones found within one model, to detect clones within other models stored in the repository.

6.1.1 Related Work

Deissenböck et al. [38, 39] present a heuristic graph-based clone detection technique for MATLAB/Simulink models integrated into the software quality framework ConQAT. The targeted MATLAB/Simulink model is imported via a model parser (*Simulink Library for Java*) that is converted into a flattened directed multi-graph with no hierarchy. Clones are detected on this flattened graph by using an extended breadth-first search in combination with a heuristic to identify isomorphic sub-graphs. Due to the use of a heuristic during clone identification, the technique does not necessarily detect all clones within a model. The detector of Deissenböck et al. is able to detect the clone types DF0, DF1 and DF2 and outputs them as a list of clone pairs.

Hummel et al. describe an index-based clone detector for the incremental detection of clones within MATLAB/Simulink models [72]. The authors index all k-subgraphs of a flattened MATLAB/Simulink model with a canonical label. This index can be updated once a model is changed. Clone groups are then derived for all k-subgraphs indexed with the same canonical label. For $k < 4$ computation is feasible even for larger models but degrades for $k \geq 4$ making it inferior to existing approaches like ConQAT.

Another clone detector for MATLAB/Simulink models has been presented by Pham et al. with the ModelCD detector in [109]. The detector is based on two separate algorithms, *eScan*, which is responsible for the detection of clone types DF0, DF1, and DF2, and the *aScan* algorithm that can be used to detect DF3 type clones. The authors use the library developed by Deissenböck et al. to convert a model file created by MATLAB/Simulink into a flattened directed multi-graph. The approach uses so-called *exas*-vectors that capture the structural properties of a node within a graph and can be used during the aScan algorithm to approximate the similarity of two nodes. Unlike the approach of Deissenböck et al., this algorithm promises completeness as it does not rely on a heuristic to find clone pairs. In a comparison between ConQAT and ModelCD, Deissenböck et al. found that due to the size of industrial size models, the approach typically does not terminate in an adequate amount of time. Clones detected by ModelCD are pruned and grouped, resulting in a set of clone groups containing all clones related to each other.

Al-Batran et al. [2] explore the detection of semantic clones within a MATLAB/Simulink model by introducing normal forms of these models. By converting a model into a normal form, semantic clones can be detected by searching for syntactically equivalent model fragments within a model. A model is converted into its normal form by successively applying transformation pattern on a model.

An approach to detect clones within MATLAB/Simulink models based on the textual representation of model files has been presented by Alafi et al. [3]. The authors created SIMONE, an adaptation of the tool NICAD [134], which is a widespread tool for code clone detection, to find clones within model files of MATLAB/Simulink. Adaptations include filtering for relevant properties, establishing a consistent order of parameters, blocks and ports, as well as renaming all identifiers to anonymize names and parameters within the textual representation. This normalizes the textual representation of all model elements within the model file and allows the text-based matching and comparison of these elements. Due to these pre-processing steps, the clone types detected by SIMONE cannot be directly mapped to the type definition proposed by Gold et al., which is also recognized by the authors. SIMONE is able to detect exact clones, renamed clones as well as near-miss clones, without considering changes to the properties of Simulink model elements like layout, non-semantic or semantic properties. The detector operates on three levels of syntactic granularity: model-, subsystem- and block-granularity, with subsystem-granularity receiving most of the authors' attention. While the approaches of Deissenböck et al. and Pham et al. focus on clones on the block granularity level, SIMONE is also able to determine if two models or subsystems are clones of each other. Clones detected by SIMONE are reported as clone pairs and directly displayed within MATLAB/Simulink. SIMONE can also be used for anti-pattern detection, by performing *cross-clone detection* against a set of pre-defined models that contain model fragments considered to represent examples of anti-patterns. During cross-clone detection, clones are detected that *cross* between two or multiple models [146].

Doerr et al. present Just Simplify [45], a pure heuristic clone detection approach that identifies potential subsystem clones in MATLAB/Simulink models based on a modified Halstead metric [65]. For each subsystem, this complexity metric is computed based on the blocks contained within the respective subsystem. Subsystems with similar complexity metrics are candidates for potential subsystem clones. Partial model clones cannot be identified by this approach. In addition, the authors sketch a process for automatic clone refactoring similar to the approach presented in this chapter but do not give detailed information about its realization or potential shortcomings.

To resolve model clones, Tran et al. [154] propose the use of atomic refactoring operations that can be used to perform model transformations on a MATLAB/Simulink model. The authors discuss several use cases for these refactoring operations, e.g. reordering ports in subsystems, union of two or more distinct subsystems or the extraction of single signals from a bus signal. Unfortunately, no approach is outlined within their work to realize clone resolution, only proposing it as future work.

The gSpan algorithm implemented by Yan et al. [164] is a graph-based pattern mining algorithm from the domain of graph mining developed to discover frequent substructures within a graph. Deissenböck et al. evaluated this algorithm on MATLAB/Simulink models and concluded that it is not suited for the use of clone detection on this type of model [38].

Chen et al. present the gApprox algorithm for the mining of frequent approximate patterns within single graphs [26]. Unlike the gSpan algorithm, which finds exact occurrences of a pattern, the gApprox algorithm further is able to find approximate patterns, i.e. pattern that maximally differ by a set amount of nodes but otherwise are the same.

Basit et al. present a mechanism for clone management based on a variant configuration language (VCL) [7]. Detected clones are converted into a VCL representation and stored within a clone repository. Subsystems generated by a VCL managed clone, are highlighted in the model. Developers can then reuse clones managed in the repository for new/other models. The refactoring procedure presented in this chapter similarly stores refactored clones within the MATLAB/Simulink library browser that can be accessed by each developer in a familiar fashion.

6.1.2 Contributions and Bibliographic Notes

The contributions of this chapter revolve around the layout-based clone detection algorithm presented in Section 6.2.1, the clone consolidation algorithm presented in Section 6.2.2 and the refactoring process for clones of type DF0, DF1 and DF2 in Section 6.3.2. The first realization of these contributions has been implemented by Stefan Schake as part of his bachelor's thesis [136] and have been published in [61].

Based on the results of the combined clone detection algorithm, we further show the feasibility of a repository guided cross clone detector building on the model repository introduced in Chapter 3.

6.2 Clone Detection Process

The clone detection process is used as a means to identify clones that can be refactored into generic library blocks as explained in Section 6.3. The overall clone detection process is shown in Figure 6.1. It allows for the analysis of models imported with the MATLAB/Simulink tool adapter (see Section 3.3.1) and supports three algorithms. The heuristic graph-based ConQAT-algorithm developed by Deissenböck et al., the approximate graph-based gApprox-algorithm developed by Chen et al. [26] and a novel layout-based copy clone detection algorithm for MATLAB/Simulink models.

In contrast to the default behavior of ConQAT and gApprox that operate on a flattened graph, we detect clones within the graphs spanned by the blocks of either individual or a pair of subsystems. By using this restriction, all clones of type DF0-DF2 detected by the applied algorithms, can be refactored into a generic library block without altering its containing environment. Except the restricting the scope of the overall clone detection algorithm, we have implemented both the ConQAT and gApprox algorithm per their specification in [26, 39]. As the adaptation of the scope only results in multiple runs of the clone detection algorithms on different subsets of the flattened model graph and does not change the semantics of the actual detection procedure, we refer to the publications related to these algorithms for further information about their inner workings. While Deissenböck et al. already evaluated gSpan, a pattern-mining algorithm, and came to the conclusion

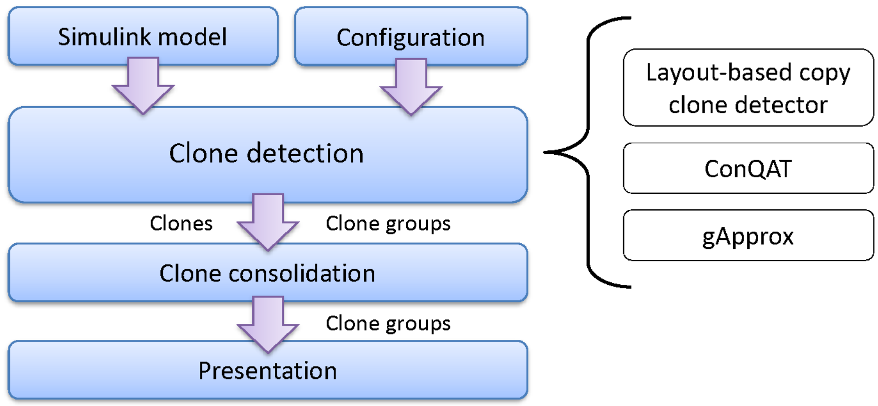


Figure 6.1: Clone detection process

that it is not suited for the application in model clone detection on MATLAB/Simulink models, gSpan was executed on a flattened graph of the MATLAB/Simulink model. As we only try to detect clones within individual or between a pair of subsystems, we evaluate the scalability limitations of the gApprox algorithm in this setting.

All clone detection algorithms try to detect one or multiple pairs of model fragments that are similar with respect to the definition of similarity used by the respective algorithm.

Definition 6.1 (Model Fragment).
A model fragment m of a MATLAB/Simulink model $\mathcal{M} = (B, P, L, I, f_p, h)$ is a connected subgraph $m = (V, E)$ such that holds:

- $V \subseteq B$
- $E = \{e = (p_1, p_2) | e \in L \wedge v_1, v_2 \in V \wedge succ(v_1, v_2) \wedge p_1 \in ports_{out}(v_1) \wedge p_2 \in ports_{out}(v_2)\}$

A clone is then defined as a pair of two similar non-overlapping model fragments.

Definition 6.2 (Clone).
A clone $c = (m_1, m_2)$ is a tuple containing two model fragments $m_1 = (V_1, E_1)$ and $m_2 = (V_2, E_2)$.

Furthermore it holds that $V_1, V_2 \neq \emptyset$, $V_1 \cap V_2 \neq \emptyset$ and both m_1 and m_2 are similar with respect to a given definition of similarity.

The actual definition of similarity varies between the used clone detection algorithm.

Both the ConQAT and the layout-based copy clone detector only detect clone pairs. Therefore, the result set of these algorithms might contain multiple clones that are similar to each other. Redundancies of this kind complicate the navigation and review of detected clones. The gApprox algorithm sorts similar clones into clone groups that contain more than two model fragments.

Definition 6.3 (Clone Group)**.**
A clone group $g = (m_b, O = \{m_1^o, ..., m_n^o\})$ is a tuple containing a model fragment m_b called the base of the clone group and a set of model fragments O containing the clone occurrences of g with respect to the clone base m_b.

Additionally it holds that $\forall\ m^o \in O : c = (m_b, m^o)$ is a valid clone.

During the clone consolidation step, we again analyze all found clones and clone groups for structural overlap and merge all clones into a consolidated result set. This result set then only contains non-overlapping clone groups that are presented to the user and can be used for clone refactoring.

Process configuration The overall detection process can further be configured by a set of options that alter the process and might lead to the exclusion of certain subsystems or subsystem pairs from the overall scope of the analysis. All of these options are activated by default.

- **Search subsystems** This option activates individual clone search within all subsystems
- **Search subsystem pairs** This option activates the aforementioned pair-wise clone search between all subsystems
- **Ignore library blocks** By activating this option, all subsystems that are either library blocks or contained in a library block are skipped during the individual and pair-wise subsystem clone search. As library blocks already represent reused model fragments, searching them for possible reusable model fragments may not be desirable in every application scenario
- **Ignore library blocks with same origin** A relaxation of the previous option is to only ignore Subsystem blocks within the same library during clone search. Clones between library blocks with different origin are still detected. This option is overridden by the previous option
- **Ignore ports** As ports are a prevalent block within each subsystem, a lot of low size clones consists of ports and commonly used blocks. This option prohibits detection algorithms from adding ports during clone search

In the following sections, we will present additional information about the layout-based copy clone detector and the clone consolidation phase.

6.2.1 Layout-Based Clone Detection

Due to the graphical user interface of MATLAB/Simulink, copy clones can be easily created through use of the copy & paste action available in the graphical editor.

Copy clones are reported to appear frequently in traditional programming for a number of reasons. Kim et al. lists language limitations as a reason for code duplication,

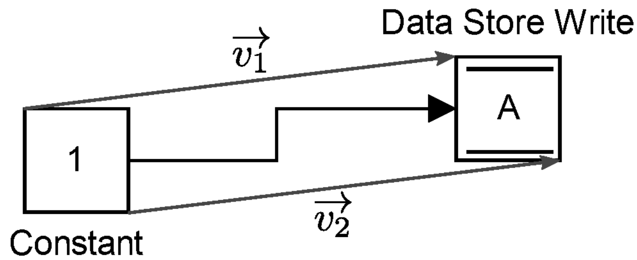


Figure 6.2: Visualization of the relative layout expressed by v_1 and v_2

if a developer cannot easily reuse code, they are more likely to use it verbatim [78]. Time constraints, insufficient knowledge, ignorance or carelessness with regard to the consequences of cloning on code maintenance and the absence of a standardized reuse process, are further reasons for the creation of copy clones given by Koschke [85]. As these reasons are rooted within the development process and the ideology of each individual developer rather than the language used, it can be expected that copy clones also are an issue in model-based development.

One property of copy clones in MATLAB/Simulink is that they typically inherit the relative layout of the initially copied model fragment. This property can be leveraged in a layout-based copy clone detector.

The relative layout of a block b can be calculated based on its position in relation to each of its adjacent blocks. The position of a block can be retrieved from the parameter *Position* stored for each block. This parameter saves four values, representing the coordinates of the upper left and lower right corner of the bounding box of b on the canvas of the hierarchy level b resides in. Based on these two points, the relative position of a block b to one of its adjacent blocks b_a can be expressed by two vectors v_1 and v_2 connecting the upper left and lower right corners of b and b_a.

Figure 6.2 shows a visualization for the vectors v_1 and v_2 expressing the relative layout between a Constant and DataStoreWrite block. The vectors v_1, v_2 not only capture the relative distance between these two blocks but also the dimensions of the respective blocks.

Definition 6.4 (Relative Layout of two Simulink Blocks).
Let b_1 be a block with position (x_{b_1}, y_{b_1}) and width w_{b_1} and height h_{b_2} and b_2 be another block b_2 with position (x_{b_2}, y_{b_2}) and width w_{b_2} and height h_{b_2}. The relative layout of b_1 and b_2 is defined as a set of vectors $l_{rel}(b_1, b_2) = \{v_1, v_2\}$ with:

- $v_1 = \langle x_{b_2} - x_{b_1}, y_{b_2} - y_{b_1} \rangle$
- $v_2 = \langle (x_{b_2} + w_{b_2}) - (x_{b_1} + w_{b_1}), (y_{b_2} + h_{b_2}) - (y_{b_1} + h_{b_1}) \rangle$

The relative layout can then be used to search for block pairs (b_1, b_2) within the input model.

Definition 6.5 (Relative Layout Clone Candidate Pair).
Let T be the distance threshold between the x- and y-values of the vectors of two relative

Algorithm 3: Pairwise breadth-first search based on block pairs

Input: Clone candidate pair *start*, Set of all clone candidate pairs *pairs*
Output: Clone as set of connected block pairs

```
clone = ∅;
queue = {start};
while queue has remaining pairs do
    dequeue element from queue as cur = (cur1, cur2);
    if cur1 ∧ cur2 marked then
        continue;
    end
    mark cur1;
    mark cur2;
    clone = clone ∪ {cur};
    b = {p = (b1, b2) ∈ pairs | succ(b1, cur1) ∧ succ(b2, cur2) ∧ b1, b2 unmarked};
    f = {p = (b1, b2) ∈ pairs | succ(cur1, b1) ∧ succ(cur2, b2) ∧ b1, b2 unmarked};
    foreach p = (b1, b2) ∈ (b ∪ f) do
        if f_cand(cur1, b1) ∧ f_cand(cur2, b2) then
            enqueue p in queue;
        end
    end
end
return clone
```

layouts. A block pair p=(b_1, b_2) is a clone candidate with respect to the relative layout if both b_1 and b_2 share the same type and have at least one successor node b'_1 and b'_2 with $\{v_1^1, v_2^1\} = l_{rel}(b_1, b'_1)$ and $\{v_1^2, v_2^2\} = l_{rel}(b_2, b'_2)$ so that holds:

- $v_1^1 - v_1^2 <= \langle T, T \rangle$
- $v_2^1 - v_2^2 <= \langle T, T \rangle$

Further we define the candidate function $f_{cand} : B \times B \rightarrow \mathbb{B}$ that returns true if the condition specified above holds for a pair of blocks (b_1, b_2) and false otherwise.

This definition can be used to identify all relative layout candidate pairs *pairs* within a given input set of blocks B_{input}. The set of identified clone candidate pairs is then used as a starting point for the identification of connected copy clones in the input set B_{input}.

Algorithm 3 shows the copy clone detection procedure in pseudo code. The approach is similar to the extended pairwise breadth-first search used by the ConQAT algorithm [39]. Starting from a candidate clone pair *start*, further candidate clone pairs are explored within the set of identified clone pairs *pairs* that can extend the relative layout clone containing the blocks of *start*. Blocks residing within a clone pair extending a found clone are marked, so that they are not considered to be part of another clone. Further clone pairs are identified based on the successor function (see Section 2.2.4) which helps to identify possible clone candidates connected via incoming or outgoing lines of the blocks of the current candidate pair *cur*.

The algorithm is repeated for each identified clone candidate pair derived from the input set B_{input} that was not part of a previous run of the algorithm. As described at the beginning of this section, due to the constraints of the refactoring procedure, we only consider clones on a subsystem level, e.g. cross clones between two subsystems or within a single subsystem. Therefore, the input set B_{input} is either the union of all blocks contained within two subsystems s_1 and s_2 — $cssys_{dir}(s_1) \cup cssys_{dir}(s_2)$ — to detect cross subsystem clones or only the set of blocks contained within a single subsystem s — $cssys_{dir}(s)$. The algorithm is executed for each possible input set B_{input} derived for each subsystem s and subsystem combination $s_1,\ s_2 \in \mathcal{P}_2(B) = \{S \subseteq B \mid |S| = 2 \wedge S \text{ only contains Subsystem blocks}\}$ of a MATLAB/Simulink model $\mathcal{M}$. The result of all runs of the algorithm is an unsorted set of copy clones that still need to be consolidated into clone groups.

6.2.2 Clone Consolidation

As the same clone might be reported by multiple runs of a single clone detection algorithm or different clone algorithms, reported clones need to be consolidated into a uniform result set. This result set shall no longer contain duplicate clones and classify reported clone pairs into clone groups. These clone groups shall contain all clone occurrences for a given clone base and shall contain subsumed clones, i.e. sub-clones that represent substructures of the clone base of the clone group. Both ConQAT and the layout-based copy clone detector only return clone pairs by construction and do not determine all occurrences of a given clone.

Clone consolidation has been realized as a two-phase process. First, we merge reported clones into clone groups to subsequently merge or remove incomplete or redundant clone groups.

Clone grouping To group clones into clone groups, we first sort all clones in ascending order based on the number of blocks they contain. After that, we compare all clones with the same size against each other. As a clone c consists of two block sets representing two similar model fragments, we check if a model fragment contained within one clone is also contained within another clone c'. If we find such a pair of clones, we create a new clone group g, set one of the found model fragments as the clone base of this group and add the remainder of the block sets as occurrences of the respective clone base. If further matches for one of the block sets contained within g are found, these are also added to the clone group. All unmatched clones are converted into simple clone groups with one clone base and one clone occurrence. Clone grouping is executed individually for each clone detector. The created groups of all detectors are consolidated during the last phase of the consolidation process.

Merge/removal of incomplete/redundant clone groups After all clones have been converted into clone groups, we again check all created clone groups for overlapping bases or occurrences analogous to the clone grouping step. If such an overlap is found, we merge the two clone groups. Furthermore, during this step we can also identify subsumed clone groups and assign them to the clone group subsuming them. A subsumed clone group is a clone group that contains subsets of model fragments of another bigger clone group, the subsuming group. As such groups may contain more occurrences than the subsuming group, we store all subsumed group within their subsuming clone group. The subsuming functionality can be deactivated as part of the configuration of the overall clone detection process.

Note that two clone groups representing the same clone structure that do not share an occurrence nor a base cannot be matched by this approach. To solve this problem, we apply a graph isomorphism check based on an implementation[1] of the VF2 graph isomorphism checker introduced by Cordella et al. [31]. By activating an option within the configuration of the overall clone detection algorithm, a graph isomorphism check is performed in addition to the overlap checks mentioned above. This might cause two clone groups with differing layout but equal structure to be merged during a run only including the layout-based copy clone detector.

Further configuration options The clone detection process can be further configured to filter certain kind of elements or clones from the result set. These options are activated by default.

- **Filter low weight clones** This option filters clones of the result set with respect to a weighting function. Currently we use the weighting function used by Deissenböck et al. in [39, Section 5.3] that weights clones based on their block type, with nonvirtual blocks having a higher weight than virtual ones
- **Check subsystem equality** When activating this option, all Subsystem blocks that are part of a clone are checked if the subsystems contained within the two model fragments are structurally the same. If this is not the case, the subsystem is not considered to be part of the clone
- **Minimum clone size** By activating this option and specifying a minimum clone size T_S, all clones with a size smaller than T_S are automatically discarded. By default, T_S is set to five as proposed by Deissenböck et al. [39]

6.2.3 Presentation

After the clone detection and consolidation phase, found clones are shown within the GUI of artshop. Figure 6.3 shows the result view of the clone detection process containing one clone group with one occurrence. The displayed clone has been detected using the

[1] `http://jgrapht.org/`

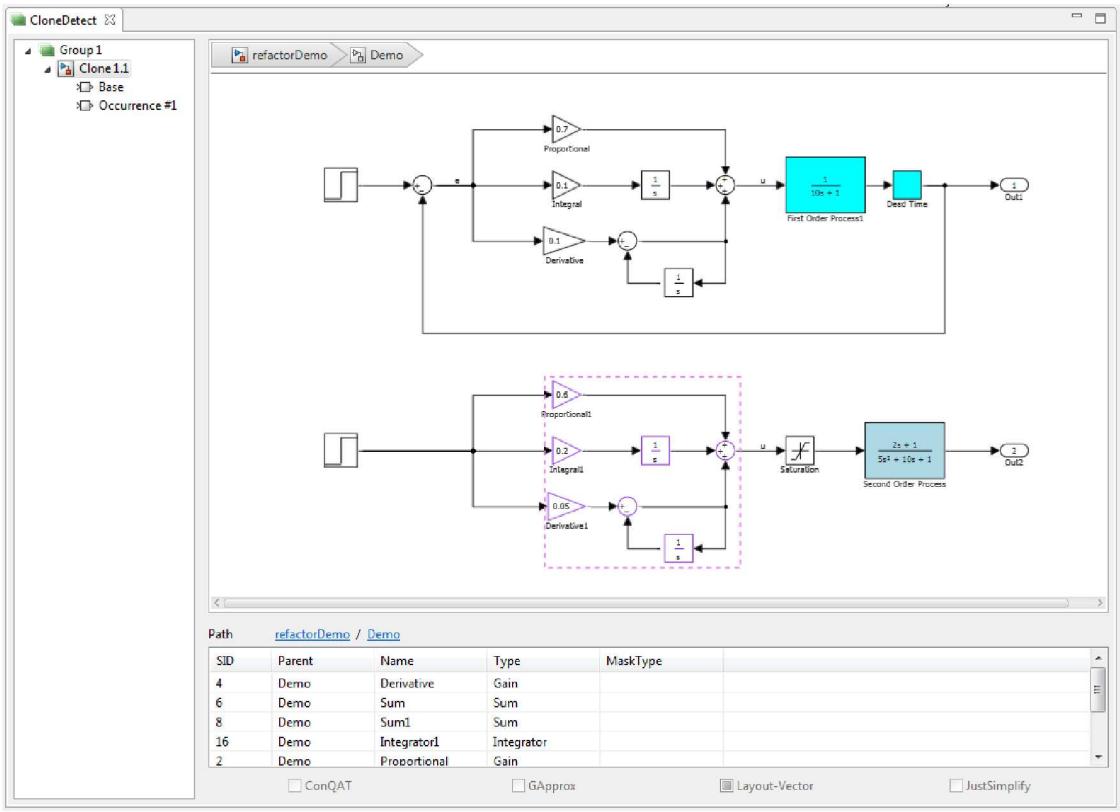


Figure 6.3: Visualization of clone groups in artshop

layout-based copy clone detector. On the left-hand side of the figure, the clone browser displays all clone groups found within the analyzed models. By selecting the base or an occurrence of the clone group, the model view automatically switches to the location of the occurrence and highlights it, as if it was selected within the view itself. In addition, the blocks of the clone occurrence are listed in the list view under the model view. Each clone occurrence can also be shown in MATLAB/Simulink itself via a context menu option within the clone browser.

Moreover, the occurrences of a clone group with clone type DF0-DF2 can be refactored by creating a generic library block for the base of the clone and replacing all clone occurrences with respective instances of this library block.

6.3 Clone Refactoring

Clones found during the clone detection process usually decrease the overall maintainability and quality of a model. Thus, it is desirable to reuse cloned functionality in a controlled manner to trace and document intra- and inter-model reuse of model fragments.

MATLAB/Simulink offers a mechanism called library blocks that can be defined by the user and inserted into arbitrary models (see Section 2.2.1). The functionality of the

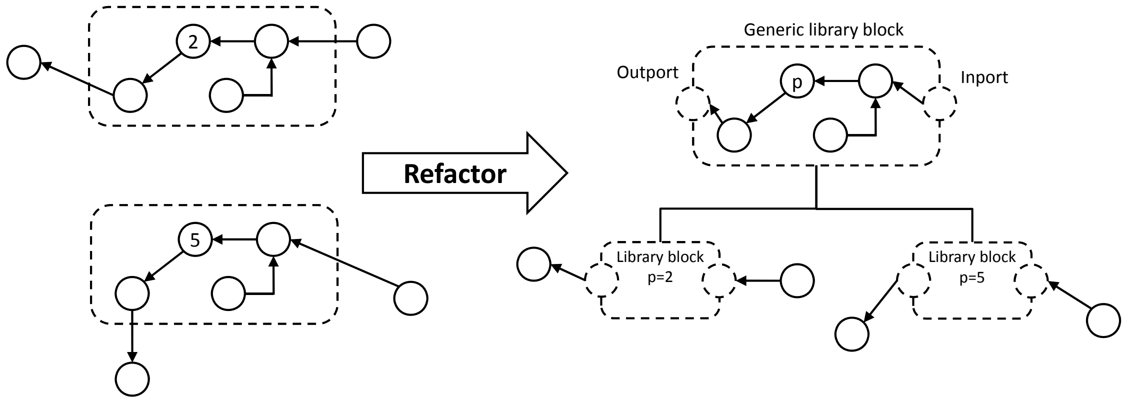


Figure 6.4: Example of the clone refactoring procedure

library block is then present within the model without the need to re-declare its content. Another advantage of library blocks is that all changes made to the actual definition of the library block are automatically inherited from the respective instances of the library block. The block is available within the library repository of MATLAB/Simulink and can be enriched with additional information, which improves the overall reuse process.

Doerr et al. [45] discuss a semi-automatic process for library block creation with subsystem granularity that only supports partial resolution of clone variations that may appear in DF0-DF2 clones. The procedure introduced in this section does not share these restrictions and can convert clones of type DF0-DF2 into generic library blocks and replace clone occurrences by correctly parametrized instances of the previously created library block.

6.3.1 Refactoring Procedure

Figure 6.4 shows a sketch of the application of the clone refactoring procedure. First, a masked library block is created based on the clone base of a selected clone group and saved within the library repository of MATLAB/Simulink. Block parameters that differ between the base and at least one occurrence of the clone group are identified and promoted. Promoted parameters can then be configured via the mask of the library block. After that, the library repository is refreshed to make the block available within the modeling environment. In the last phase, the clone base and all its occurrences are replaced by a parametrized instantiated version of the created library block. Note that only clone occurrences need to be parametrized, as the default configuration is derived from the clone base.

Creation of the library block The first step during the clone refactoring procedure is to create a library block that will contain the duplicated model fragment. After the creation of the library block, all blocks and lines contained within the clone base are recreated within the library block. For each line connecting the blocks of the clone base with the rest of the model, a corresponding in-/outport is created, representing the interface of the clone base. A port mapping m_p is saved that maps the in- and outports of the library blocks to the in- and outports of the respective blocks within the library block. In addition, the Subsystem block representing the library block is masked and user-defined descriptions and documentation may be added to the mask dialog. A documentation block placed within the library may contain further documentation or change logs of the library block. The created library block can then be placed within an existing block library or a new block library can be created and integrated into MATLAB/Simulink.

Determine parameter differences For DF1 and DF2 clone groups, the parameter differences between the clone base and the clone occurrences need to be determined. First, we determine all block types of the clone group and the parameters that can be configured by the user to change the behavior of a block with a given type. To calculate the differences between a block b_o in a clone occurrence against its corresponding block b_b within the clone base we use an injective mapping function m_i. This mapping function maps the blocks of the clone base to their respective counterpart in the i-th clone occurrence and is created during the clone detection procedure. Determined differences between the clone occurrences are stored and for each detected differing block parameter of a given block b_b in the clone base, a mask parameter is created in the mask of the library block. The block parameter of the block b_l corresponding to a differing parameter of block b_b within the clone base is promoted into the mask of the created library block. This enables the configuration of this block parameter within the mask of the created library block. By default, this parameter is set to the value used in the clone base, but may be configured differently during clone instantiation.

Refresh library repository Before an instance of the new library block can be created, the library repository of MATLAB/Simulink is refreshed to make the library block available within the modeling environment.

Instantiate library block The library block created in the previous phases can now be used to replace the model fragments of the clone base and the individual clone occurrences within the model. First, a mapping of the environment m_e of each model fragment is created that stores how blocks of the clone occurrence are connected to blocks of their environment including individual blocks and their connected ports. After that, all lines are deleted that start/end at least one block contained in the model fragment. Furthermore, all blocks within the model fragment are replaced by an instance of the created library block. The library block is connected to its environment based on the previously stored mapping m_e that is again applied to the in- and outports of the library block using functions m_i and m_p.

After the library block is created and connected for all model fragments of the clone group, the individual mask parameters are set to the values identified in the second phase.

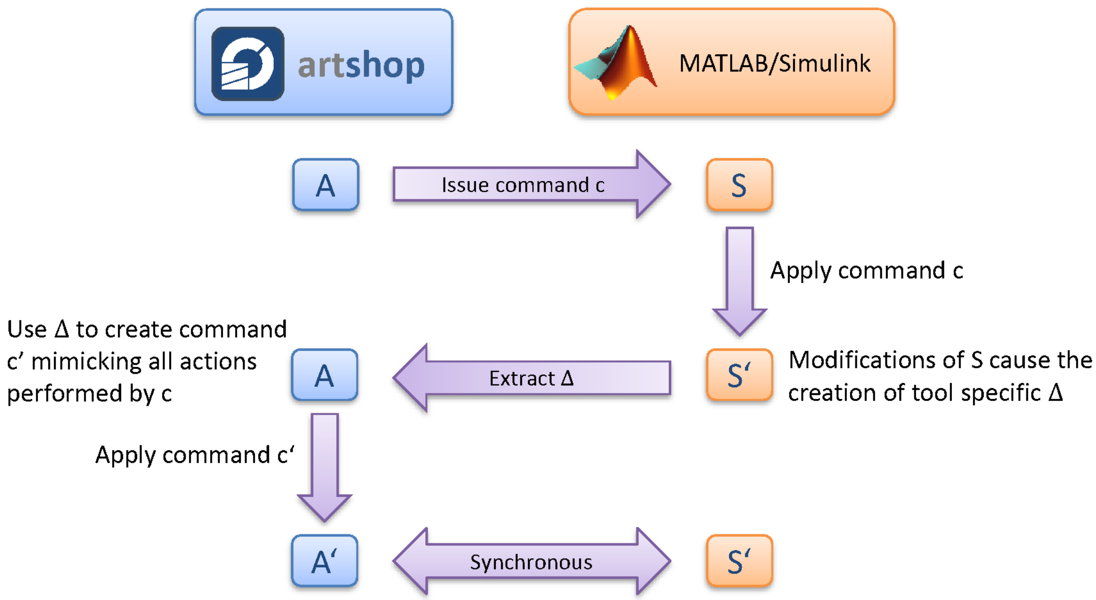


Figure 6.5: Synchronous model transformation in artshop and MATLAB/Simulink

6.3.2 Modification Commands

The previous section described the general process of refactoring a clone group by using parameterized library blocks. As already proposed by Tran et al. [153, 154], atomic model transformation operations can be composed to realize complex refactoring operations. While the techniques discussed by Tran et al. are directly applicable in MATLAB/Simulink, we actually need to transform two models during the refactoring procedure: the MATLAB/Simulink model and its corresponding representation loaded in the artshop framework. A naive method would only refactor the MATLAB/Simulink model
procedure described in Section 3.5. As this requires a complete reimport of the model that, depending on its complexity, could take a large amount of time, we instead opted to refactor the artshop model representation simultaneously.

To realize this synchronous transformation, we first implemented atomic modification commands that are able to transform the actual MATLAB/Simulink model according to the refactoring procedure outlined in the previous section. Examples for modification commands are: *SetParameter*, *AddLine*, *AddBlock*, *AddLibrarySection*, *SetDocumentation*, *DeleteLine*, *DeleteBlock*, *PromoteMaskParameter* and *CreateBlockMask*. We emulated the command architecture of the EMF [145] to implement these commands and used utility functions implemented as part of the MATLAB/Simulink tool adapter to realize the transformation in MATLAB/Simulink.

To apply the specified refactoring commands in a synchronous fashion, we mimic the transformation performed by the commands transforming the MATLAB/Simulink model by an equivalent command changing the artshop model representation as shown in Figure 6.5. As some commands like the *AddBlock* command introduce a tool specific Δ of information within a created element, like block parameters or the SID, we need to extract this Δ from MATLAB/Simulink to represent a created element in artshop correctly. The Δ is subsequently used to create a block and add all extracted parameters contained within the Δ to the block, so that they are again available in the graphical viewer and the analyses of the framework.

Another case that needs to be considered during the deletion of blocks and lines, is the prevention of dangling references within the artshop model representation. As other elements not considered during the refactoring procedure may reference a deleted block or line in the artshop model, a cross reference analysis is performed to remove dangling references to deleted elements. During this analysis, all references to the deleted element are also removed as part of the deletion command, considering all elements within the repository as well as all elements not yet committed.

This implementation of the refactoring process ensures that both the MATLAB/Simulink model and the artshop model representation are synchronized to each other once the refactoring procedure is finished.

6.3.3 Limitations

While the refactoring procedure is able to handle clones of type DF0-DF2 there still exist certain limitations in its application. Unlike the refactoring procedure described by Doerr et al. [45] the refactoring procedure presented in this thesis can also be used to refactor only a subset of blocks within a subsystem. We derive the interface of the created library block from the clone base based on the lines that connect the blocks of the clone base with its environment. Here, it might be the case that additional outgoing connections exists within one or multiple clone occurrences that are currently not accounted for. Extending the interface to include all outports required by all clone occurrences would unnecessarily bloat the interface of the library block and result in multiple, possibly redundant ports, which are only used by a subset of clone occurrences. Further work needs to be invested to correctly handle these cases or disable the refactoring procedure altogether for clone occurrences with more connections to the corresponding environment than the clone base.

6.4 Evaluation

We now present the evaluation results of the techniques related to the layout-based clone detector and the clone group consolidation procedure. The evaluation has been performed on the same system presented in Section 3.5 and with the models introduced in Section 3.5.1. While we also implemented the ConQAT and gApprox algorithm, the evaluation of these techniques here is accessory, as no contributions have been made towards improving them besides the restrictions applied as part of the overall clone detection process (see beginning of Section 6.2).

Table 6.1: Performance and results of the clone detection algorithms

Model	LCCD t (ms)	LCCD Groups	LCCD Clones	ConQAT t (ms)	ConQAT G.	ConQAT C.	gApprox t (ms)	gApprox G.	gApprox C.
MAV	184.71	3	53	186.70	9	67	1807.19	11	72
DAS	323.22	4	9	146.78	10	21	4615.54	13	29
EL	426.60	9	31	233.77	19	53	17006.75	30	73
PI	11032.07	14	358	14758.92	13	151	n/a	n/a	n/a
ECLA	432.91	4	9	481.94	22	46	4591.99	70	178

We first evaluate the performance of the implemented copy clone detector and the clone consolidation procedure. After that, we examine the overall quality of clones detected with the copy clone detector in comparison to the other approaches.

6.4.1 Performance

To evaluate the performance of the layout-based copy clone detector (LCCD), we executed each clone detector individually on each evaluation model and measured the computation time, obtained clone groups and the absolute amount of clones within all groups. During the evaluation, we used the default configuration of the clone detection procedure. Moreover, the gApprox algorithm was configured to skip the detection of DF3 clones, to highlight the effect of the clone consolidation procedure.

Table 6.1 shows the empirical evaluation results of the performance evaluation. The amount of reported clone groups and contained clone occurrences correspond to the top-level clone groups and their clone occurrences. Subsumed clone groups are not part of this statistics. Unfortunately, our implementation of the gApprox algorithm did not terminate on the PI model, as it found a huge clone between two subsystem both containing around 100 blocks, leading to the algorithm running out of memory during its resolution. This partially confirms the results observed by Deissenböck et al. with regard to the application of pattern mining techniques even in our restricted application scenario, as mentioned at the beginning of Section 6.2 [38].

Comparing the LCCD against the other approaches, it becomes apparent that it has a similar run-time as the ConQAT algorithm while detecting substantially fewer clone groups. Nevertheless, copy clones could be found in every model, highlighting the importance of the algorithm. In comparison to the gApprox algorithm, only a minor subset of the overall clones detected are copy clones. Many clones detected by gApprox are clones resulting from the use of structural blocks, i.e. virtual blocks.

It is noticeable that the time to compute the clone groups on the PI model is significantly higher than for the other evaluated models, particularly in comparison to the structurally similar ECLA model. In contrast to the PI model, the ECLA model consists mostly of library blocks, which are ignored by the clone detection algorithm. Therefore, all algorithms perform much faster on this model than the PI model. In addition, the PI

Table 6.2: Performance and results of the clone consolidation procedure

Model	Before merge		t_{merge}	After merge			
	Groups	Clones	(ms)	Groups	Clones	Sub. groups	Sub. clones
MAV	23	192	55.57	11	73	10	63
DAS	27	59	23.30	16	35	3	11
EL	58	157	80.80	30	76	43	228
PI*	27	509	164.56	16	167	36	727
ECLA	96	234	90.02	70	192	54	230

* Results for the PI model do not contain the results of the gApprox algorithm.

model has the highest amount of subsystems of all evaluated model (1038), some of which contain up to 98 blocks with only few library blocks. Each of the clone detection runs shown in Table 6.1 were executed with only one active clone detection algorithm. Thus, the removal and merging of redundant clones was only performed on the result set of one algorithm.

Table 6.2 shows how the clone consolidation procedure reduces the amount clone groups if multiple clone detectors are active during the clone detection procedure. The first two columns show the accumulated amount of clone groups obtained from the individual runs of the clone detectors shown in Table 6.1. Applying the clone consolidation procedure on the individual results of the used clone detectors significantly reduces the amount of clone groups and clones within the result set, as shown in the third and fourth column of the table. Redundant clones are merged and subsuming clones across the results of the individual detectors are detected and stored accordingly. The number of subsumed clone groups and their contained clone occurrences are listed in columns five and six. On average, the clone consolidation procedure reduces the amount of clone groups by 41.80 % and the amount of clone occurrences by 47.87 % within a maximal processing time of around 200 ms. Therefore, the consolidation procedure is a useful tool during the manual assessment of clone groups after the analysis is finished.

Consider Figure 6.6 that displays the overlap between the individual results of all three algorithms for the EL model. It shows that the results of the LCCD were also found by all other algorithms. Both the ConQAT and the gApprox algorithm provide distinct clone groups alongside groups that are unique to these two. We further note that the amount of clone occurrences of the gApprox algorithm does not add up to the value shown in Table 6.1. The four missing clone groups were merged with one of the results of the ConQAT algorithm that provided an occurrence that overlaps with the occurrences of the gApprox algorithm.

6.4.2 Quality

In the previous section, we already discussed the amount of occurrences found by the LCCD in comparison to the ConQAT and gApprox detectors. To further reason about the overall quality of the results sets of each individual algorithms, we use a weighting

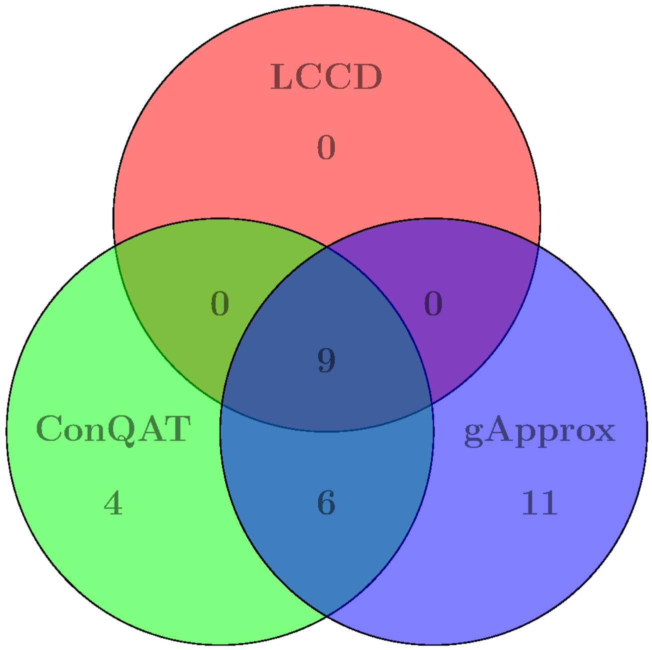


Figure 6.6: Overlap detected between the clone groups detected by the clone detection algorithms applied on the EL model

scheme introduced by Deissenböck et al. [39, p. 6, last paragraph]. This scheme assigns a weight to each block contained within the base of a clone group and sums up all weights of the blocks of a given clone base. Nonvirtual blocks have a rating of 1 or higher while virtual blocks are typically rated with 0, which favors clone groups impacting the semantic behavior of a model.

Table 6.3 shows the amount of clone groups, their containing clones, the average size of the base of a clone and the weight calculated using the mentioned weighting scheme for each algorithm on the set of evaluation models. Clones detected by the LCCD on average have the highest weight across all detectors, highlighting the quality of detected clone groups. The high weight is related to the low amount of detected clone groups in

Table 6.3: Quality of detected clones

Algorithm	Groups (Clones)	⌀Size	⌀Weight
LCCD	34 (460)	9.03	14.40
ConQAT	73 (339)	10.94	12.27
gApprox*	124 (352)	7.33	8.88
Merged	143 (543)	9.32	10.44

* Results of the gApprox algorithm do not contain the results of the PI model.

Table 6.4: Performance evaluation of the refactoring procedure for the clone group shown in Figure 6.3

Phase	t (ms)	Executed commands
Create library block	1308	29
Refresh library browser	5294	1
Instantiate clone base	6626	6
Instantiate clone occurrence	4303	7
Total	17531	43

comparison to the other algorithms that do not rely on hints, i.e. layout information, to compute their result sets. Furthermore, the results of the LCCD are often directly found or even subsumed by the result of other detectors. Nevertheless, the LCCD can be applied to identify copy clones within a MATLAB/Simulink model and might even be used as a clone group classifier, to assign clone groups containing copy clones a higher relevance, similar to the weighting scheme of Deissenböck et al. Merging the clone groups of all detectors does not significantly decrease the overall quality even if lower quality clones detected by the gApprox algorithm are also included within the result set.

6.4.3 Refactoring Procedure

To give an impression of the overall performance of the refactoring procedure, we analyzed the refactoring of the example clone displayed in Figure 6.3. Table 6.4 shows the time needed for the phases of the refactoring procedure as well as the amount of commands executed during each phase. The most expensive phase, with respect to the time needed to execute it, is the refresh library phase, as MATLAB/Simulink needs to reload all registered library blocks. The instantiation of the clone base is slightly more computationally expensive as it also includes the loading time of the model containing the model fragments that need to be refactored. An additional command is needed to instantiate the clone occurrence, as the mask parameters of its library block instance need to be set to preserve the semantic behavior of the replaced model fragment. In comparison to the creation of the library block, instantiating it takes much more time as within these phases the artshop model representation also has to be altered. For each deleted model element, a cross reference analysis has to be performed to prevent dangling references from other objects to the deleted blocks and lines.

Overall, the clone refactoring procedure takes significantly less time than manually replacing the affected model fragments, while simultaneously synchronizing the artshop model representation with the changes made outside of the framework.

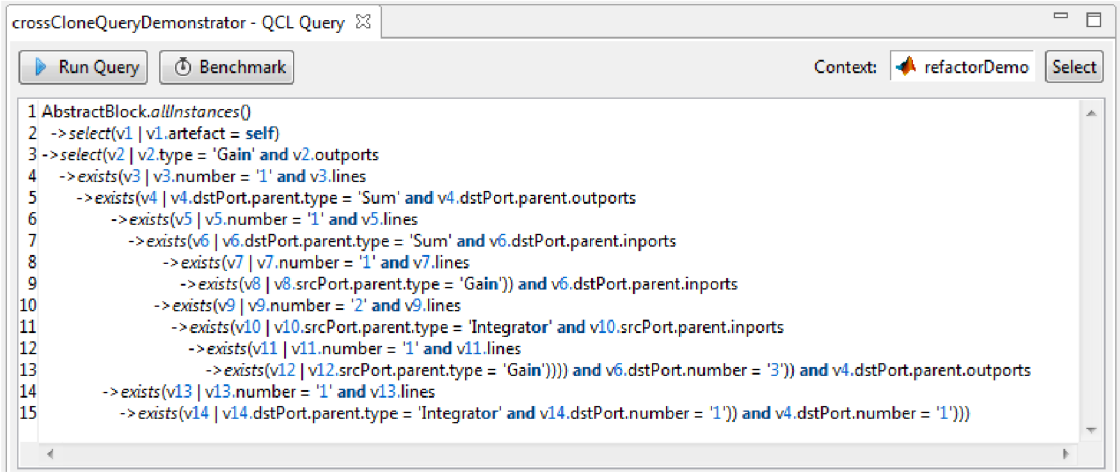


Figure 6.7: Generated query for the clone group shown in Figure 6.3

6.5 Repository Guided Cross-Clone Detection

Clones identified within one model may also appear in other models that were not part of the scope of an initial clone detection run. Depending on the size of these models, running the complete clone detection procedure again might be inefficient and possibly not even result in the same clones, if it only contains a single occurrence of a previously identified clone group. Therefore, it would be desirable to search for occurrences of already detected clones in other models to replace these occurrences with the same generic library block.

We have implemented a proof-of-concept cross-clone detector that can identify clones found in an arbitrary model $\mathcal{M}$ in another model $\mathcal{M}'$ stored within the artshop repository. Clones from $\mathcal{M}$ within $\mathcal{M}'$ can be identified based on their structural composition, i.e. for a given cloned model fragment m from $\mathcal{M}$ we search for an isomorph model fragment m' in $\mathcal{M}'$ with the mapping f_i expressing the isomorphism between m and m'. The model fragment m' is an occurrence of m within $\mathcal{M}'$ if holds:

- $\forall\ v \in V : f_i(v)$ has the same block type as v
- $\forall\ e \in E : f_i(e)$ starts and ends at an in- and outport with the same numbers as the ports connected by e

To identify occurrences of an identified clone group we utilize the querying capabilities of the artshop model repository. Given a model fragment $m = (V, E)$ from a clone group, we create an OCL query that queries an arbitrary functionmodel f for a block b_o matching the type of a block b_m from m. The query enforces that for each block $b \in V$ that is connected to b_m via a line $l \in E$, there exists a block b' connected to b_o via a line l' with the same characteristics (starts/ends at an in- and outport with same number) as l. These properties are checked for each block contained in m.

Figure 6.7 shows the query generated for the clone group shown in Figure 6.3. Each query receives a context as a parameter; within the example this is the imported functionmodel *refactorDemo* that is referred to by the *self* expression. The query starts by selecting all blocks registered in the repository in line 1 that are contained within the functionmodel selected as the context of the query. Starting from an arbitrary block contained within the clone base of the selected clone group, we encode a depth-first search within the query. The query searches for a connected component within the set of selected blocks that matches the block types and lines of the selected clone group, as described in the previous paragraph.

We performed a short benchmark on the overall performance of the proposed approach against a Java based pattern detection algorithm with the example query shown in Figure 6.7 on the ECLA model introduced in Table 3.4. The server on average took 91 ms to evaluate the query against the ECLA model without finding any occurrences of the specified model clone. While searching the same occurrence within a locally loaded instance of the ECLA model with the Java based pattern detector only took around 12 ms, it took around 10 seconds to lazily load the complete model from the repository to perform the local analysis. Therefore, the repository-based analysis is well-suited to check one or multiple models stored within the repository for the occurrence of locally detected cloned model fragments without loading them into the memory of the client machine. Further work is needed to integrate this method into the framework and allow the storage of found/refactored clones to search for occurrences in other models at a later point in time. Directly loading a library block into the query engine to automatically search for matching candidates for this block in other models might also enhance the workflow during the reuse of model fragments.

6.6 Conclusion and Future Work

In this chapter, we proposed a clone detection process that integrates the results of multiple individual clone detection algorithms and exploits their different capabilities to yield a uniform and consolidated result set refined for inspection by a modeling expert. This result set consists of clone groups, containing one clone base and a set of clone occurrences representing clones of the respective clone base. Besides two algorithms from the literature, ConQAT and gApprox, we further introduce a novel layout-based copy clone detector. This algorithm detects copy clones based on the relative position of their elements within the model in addition to structural properties, e.g. block type or connectivity, used by gApprox and ConQAT. The consolidation procedure merges the individual result of all algorithms by combining duplicate or redundant clone groups detected by multiple algorithms and hierarchically subordinate subsumed clone groups into their containing clone groups, leading to a significant reduction of around 40 % of top-level clones, while being computationally inexpensive. Computed clone groups can be browsed and are visualized using the graphical viewer provided by the MATLAB/Simulink tool adapter. We additionally propose a refactoring process for detected clones of clone type DF0-DF2 by replacing all clone occurrences with a generic library block matching the structure of the clone. This library block can be further parameterized to match the behavior of the respective clone occurrence.

The evaluation showed that the results of the layout-based copy clone detector had the overall highest quality across all clone detectors, but also detected far less clone groups than the other approaches. In addition, most results of the layout-based copy clone detector were also found by the other detectors. Nevertheless, copy clones were found in every model within the set of evaluation models. To exploit the relative layout property used by this detector further, the relative layout could be incorporated into a weighting scheme similar to the ConQAT weighting scheme described in Section 6.4.2. As a typical problem during clone detection is the sheer amount of clones detected by the detectors [38], it is reasonable to prioritize copy clones over other clones as part of clone prioritization.

In future work, the views displaying the results of the detection could be extended. Currently, clones can be navigated via the tree spanned by the hierarchical composition of clone groups and their subsumed groups. The order of the top-level clone groups is currently based on the order of detection and does not incorporate weighting schemes to order clone groups by relevance. Multiple weights as proposed by Deissenböck et al. in [38, 39] or a weighting function based on the relative layout of contained model fragments could be used to order clone groups by relevance. Furthermore, certain use-cases might require a flat-view of detected clone groups instead of the tree of subsumed clones.

As part of future work, the limitations of the refactoring procedure that have been discussed in Section 6.3.3 concerning the clone interface to its environment need to be addressed. This can be achieved by either extending the procedure to compute the maximal interface over all clone occurrences including the clone base or by prohibiting the refactoring procedure on clone groups inheriting the discussed property.

7 Model Smell Detection in MATLAB/Simulink Models

In the previous chapter, we have described a design flaw of model-based software namely the presence of multiple occurrences of the same model structure within a MATLAB/Simulink model. This quality deficit can also be present in traditional software development artifacts in the form of duplicate code within a given code base. Duplicate code is one of the quality deficits described as code smells introduced by Fowler [54] as a way to identify program locations that could benefit from being refactored or restructured. Code smells can be seen as symptoms for the violation of fundamental design principles, which negatively affect the quality of the software containing them [149]. Typically, such locations are also said to contain technical debt, which describes the debt that accrues when a developer knowingly or unknowingly makes a wrong or non-optimal design decision [149]. This term was first discussed in an article by Cunningham in [32] as a metaphor for poor technical realization of software. Refactoring code smells is a way to decrease the technical debt and at the same time increase the quality of the software. As we have already introduced the detection and refactoring of clones in MATLAB/Simulink models in the previous chapter, investigating further mappings from traditional code smell detection into the domain of model-based software development seems promising.

7.1 Overview and Outline

In this chapter, we apply the concept of code smells from traditional software development to MATLAB/Simulink models in model-based software development by introducing a catalog of model smells. These smells are partially derived from the set of code smells defined by Fowler [54] but further include smells that are unique with regard to certain characteristics of MATLAB/Simulink models. A smell might be introduced by changing a feature with insufficient knowledge of a model, or by realizing features/bug fixes under tight deadlines and introducing technical depth. In contrast to invalidated model guidelines or conformity rules, a model smell does not necessarily indicate a syntactic/semantic problem or error, but rather a weakness in the design of the system, which negatively influences the quality of the model. If a smell is detected in a model, the developer has to decide individually if the smell represents an actual problem or not.

In the following, we will first discuss related work in the area of quality assessment for MATLAB/Simulink models and describe the contributions of this chapter. After the introduction of the model smell catalog in Section 7.2, we will describe the detection strategies of these model smells in Section 7.3 including their realization in the artshop framework. We show the relevance of the defined model smells and evaluate the performance of our detector on models from academic and industrial case studies in Section 7.4 and conclude our findings in Section 7.5.

7.1.1 Related Work

The term model smell also appears in the work of Arendt et al. [6], where the smells introduced by Fowler [54] are applied to models from the Eclipse Modeling Framework [145].

In the work of Stephan et al. [146], the clone detection tool *Simone* [3] is used as the foundation for a tool called *SIMAID*, which is able to detect anti-pattern in MATLAB/Simulink models. The authors use the clone detection framework to find occurrences and approximations of pre-defined anti-pattern via cross-clone detection. One of the pattern defined by the authors is called primitive obsession, which occurs when a subsystem only contributes very simple or primitive operations. Another pattern is the feature envy pattern that occurs if the functionality of a subsystem is focused on another subsystem it contains. Patterns are modeled in MATLAB/Simulink and handed to SIMAID for cross-clone detection. While the approach has the advantage that each Simulink modeler has the tools to model an anti-pattern, the expressiveness is limited to unwanted structural attributes of a model, such as functional decomposition and block sequences. In particular, architectural smells regarding the signal structure of a model cannot be expressed with these templates.

The work of Hu et al. [71] focuses on the definition of a quality model for MATLAB/Simulink models based on the ISO/IEC 9126 [74]. The authors focus on the quality metrics targeting the internal quality of a model and introduce metrics to measure quality characteristics of their defined quality model. These metrics include simple metrics that represent the amount of certain types of model elements as well as metrics that require static analyses, e.g. model slicing.

Scheible et al. [138, 139, 140, 141], follow a similar approach as Hu et al. The authors also define a quality model based on the model proposed by Cavano & McCall [25] and relate different metrics to quality criteria and characteristics to measure the quality of a model. A metric measuring the maintainability of a model would be the 'Average signal length' metric that is categorized by Scheible as a metric of medium influence on the quality of a model. The metrics cover syntactic, structural and visual aspects of the analyzed Simulink model. Unlike in the work of Hu et al., metrics are calculated and normed to a range between 0 % and 100%. The resulting values have to be above a defined minimal/maximal value or in a specified interval to satisfy a defined quality criterion. These thresholds can be calculated based on a high quality reference model. The normed results of all calculated metrics are then aggregated in a single quality rating of a MATLAB/Simulink model.

Dajsuren et al. and Olszewska et al. present complexity and modularity metrics that aim to provide information about the overall quality of a MATLAB/Simulink model in [36, 104, 105]. They are defined on a structural level and include metrics derived from common code metrics such as the Halsted metric [65], Cyclomatic complexity by McCabe

[93], the Fan-In/Fan-Out metric by Henry and Kafura [69] or structural and data flow complexity metrics inspired by Card and Glass [24]. In contrast to the complexity metrics presented in Section 5.7.1, these metrics do not include semantic dependencies of elements within the model in their calculation.

Farkas et al. [49, 50, 51], present a framework for the evaluation of user-defined conformity rules specified with the Query-Based Rule Description Language (QRDL) on MATLAB/Simulink models. The framework has been created during the MESA project [51] and is able to check these rules on models imported from MATLAB/Simulink and IBM Rational DOORS. The functionality is now included in the tool *Assessment Studio*.

Other approaches featuring the validation of conformity rules are the MATE project [148], the Matlab Model Advisor (MMA) [128] and the commercial tool MXAM [123]. These approaches can validate pre- and user-defined conformity rules, with MATE also supporting semi-automatic repair actions that can be applied on model parts invalidating certain rules. An example for a set of rules realized by the MMA are the modeling guidelines created by the MathWorks Automotive Advisory Board [131], which were created by industrial partners in cooperation with MathWorks. All approaches are directly integrated into MATLAB/Simulink.

Kemman et al. present the *INProVE* framework (Indicator-based Non-functional Property-oriented eValuation and Evolution of software design models) that is able to check user-defined quality characteristics on different kinds of data-flow models such as MATLAB/Simulink, Ascet, Modelica or Labview [77]. Checked characteristics can be created based on pre-defined quality indicators and refined by the user by combining and aggregating different quality indicators with each other.

A general-purpose tool used to specify invariants on instances of an UML or MOF model is the Object Constraint Language (OCL) [158]. With the OCL, invariants, rules and constraints can be defined using a textual language that cannot be expressed by diagrammatic notations. An OCL statement is composed of four parts: a context defining the class the statement should be applied to, a property of the context, an operation that further qualifies this property and a conditional expression that checks if the specified statement holds in the given context. The OCL contains a rich-set of operations on various kinds of collections, which eases the querying of qualified properties. Applying an OCL statement to a valid model instance is side effect free, meaning that checking a statement will not result in changes on the model. As the OCL can be applied on arbitrary models specified based on the UML, MOF or EMOF metamodel, it can be used in any environment that uses models based on these respective metamodels for the specification of user-defined conformance rules.

To resolve detected violations of modeling guidelines, conformity rules, anti-pattern or model smells, refactoring operations can be used to transform malformed model structures automatically. Tran et al. [153, 154] describe refactoring operations for MATLAB/Simulink models that can be used to restructure a model by creating/resolving subsystems, altering/resolving bus signals or even resolving indirect signal flows by directly routing signals through the model without the need of Goto/From blocks. The approach automates complex and repetitive workflows that frequently occur when altering complex Simulink models and therefore reduces the amount of time needed to perform changes as well as the amount of errors during manual model restructuring.

Klauske et al. [79, 80] propose a method to automatically layout the content of subsystems within a Simulink model by a specifically tailored layout algorithm. In combination with other refactoring operations such as the ones presented by [148, 153, 154] it can be used to clean up the visual representation of a subsystem after model elements have been created, moved or altered.

Mengi et al. [94] propose to use refactoring operations to restructure a model to create different variants by the definition of an underlying software product line. The authors extract a commonality and difference model from existing variants. These are subsequently used to derive variation points. Variants can be created by combining the commonality and difference model via the application of the respective refactoring operations.

7.1.2 Contributions and Bibliographic Notes

The model smells presented in this chapter have been developed in cooperation with Quang Minh-Tran (DCAITI) and Christian Dziobek (Daimler) and have been published in [60]. The first version of the model smell detector has been created as part of the bachelor's thesis of Dennis Weir [159] and was later extended by Mirko Kugelmeier and the author of this thesis. We have implemented the actual detection rules presented in Section 7.3 in the artshop framework, while the resolution operations have been developed by Quang Minh-Tran and are described in [153, 154].

7.2 Model Smells for MATLAB/Simulink Models

During the development of MATLAB/Simulink models, certain model properties need to be designed with care to achieve high quality models, as it is the case in traditional software development. The naming scheme of elements plays an important role as part of the documentation of a model. Architectural properties are altered by the overall structure of the model with respect to the decomposition of functions via Subsystem blocks. Moreover, the interfaces of Subsystem blocks provide hints regarding the implementation of functionality encapsulated in subsumed hierarchy layers. Further architectural properties are related to the signal flow of a model, i.e. how signals are exchanged in a model, and the structure of bus signals.

We define a set of quality anti-patterns, called *model smells* that have been gathered in cooperation with developers from Daimler and target the aforementioned properties. These smells are partially inspired by the code smells presented by Fowler in [54] and requirements stated by certain norms and standards as the ISO 26262 [73]. A model smell contains a set of prerequisites a particular set of model elements has to fulfill in order to become an instance of the model smell. These prerequisites may target properties of individual elements or their environment but may also target dependence relations among these elements. Only if all prerequisites are satisfied, a model smell is instantiated for a set of model elements.

In the following sections, the individual model properties and their accompanying model smells are explained in more detail. We will use the model formalization introduced in Section 2.2.4 and the formalization of signal flow from Section 5.3 for the definition of the various model smells.

7.2.1 Naming Conventions

Naming model elements is an important factor influencing the overall maintainability, readability and understandability of a model. The terminology used when naming elements should come from the domain the model is used in, as the same terminology might have different meanings in two distinct domains. The name of a model element provides additional information about the use of an element, i.e. the name of a Subsystem block might provide meaningful data regarding its contained functionality. When correctly chosen, the name of an element can provide useful information regarding model documentation, re-usability and understandability of the element within its context.

Therefore, choosing a vague name for a model element is detrimental to the overall model quality, as further inspections are needed to understand the role of the model element.

Definition 7.1 (Vague Name)**.**
A model element is said to be vaguely named, if its name does not relate to the functionality it represents.

As stated in the ISO 26262 [73, Part 6 - p. 14 Table 1 - Item 1h], all model elements need to be named reasonably to increase the understandability of a model.

Another smell related to naming elements is connected to the representation of the interface of a subsystem block by the In-/Outport blocks in the subsystem.

Definition 7.2 (Inconsistent Port Block Name)**.**
The name of an In-/Outport block is said to be inconsistent*, if it differs from the name of the signal that is propagated over it.*

By semantically connecting the name of a port block to the signal that should be propagated over it, the interface of its containing subsystem or even functionmodel becomes more explicit and easier to use, as the interface describes the signals it expects to receive.

7.2.2 Partitioning

The concept of decomposition of functionality and the introduction of abstraction levels is a common principle in model-based development. Not abiding or excessively using this concept can again have a negative impact on the understandability and extendability of a MATLAB/Simulink model. Typically, a subsystem should realize a single well-defined function corresponding to the abstraction level it is placed at or contain subsystems that further decompose a given functionality. One example of excessively applying the concept of functional decomposition are reflected in the model smell *Superfluous Subsystem*

Definition 7.3 (Superfluous Subsystem)**.**
A subsystem is called a superfluous subsystem if the functionality it encapsulates is to simple and reduces the readability of its containing subsystem.

In the context of a subsystem, superfluousness can be defined by the type of blocks a subsystem contains, e.g. only virtual blocks, by a threshold representing the minimal amount of blocks a subsystem has to contain or even by a complexity metric such as the Halstead [65] or Cyclomatic complexity metric [93].

The opposite of a superfluous subsystem is a subsystem that realizes more than one function without further decomposition of these functions into separate subsystems.

Definition 7.4 (Subsystem with Multiple Functions)**.**
A subsystem realizing more than one function without the decomposition of these functions into further subsystems, is said to possess the Subsystem with Multiple Functions model smell.

This smell is similar to the code smell *Long method* introduced by Fowler [54], which describes an overly long method that should be refactored into smaller subroutines. Decomposing a subsystem possessing this smell into further subsystems named according to their realizing functionality, increases the readability of a subsystem, as the naming scheme might make further inspection of an appropriately named subsystem unnecessary.

Another example of excessively introducing abstraction layers in a model are reflected by a model containing to many hierarchy levels in the form of nested subsystem blocks.

Definition 7.5 (Deeply Nested Subsystem Hierarchy)**.**
A subsystem that is nested in more than T_{Level} subsystems is considered to be nested to deeply within the models hierarchy, with $T_{Level} \in \mathbb{N}$ being the maximum amount of acceptable hierarchy levels.

Introducing to many hierarchy levels in a model degrades the readability, as relevant model parts might be spread across many hierarchy levels. Depending on the value of T_{Level}, eliminating occurrences of this smell might increase the number of occurrences of the *Subsystem with Multiple Functions* smell. As the occurrence of these smells simply are hints for potential weak design, the value of T_{Level} can be tweaked to remove these contradicting occurrences or their occurrence could be documented with a proper reasoning.

As we have seen, functional decomposition has to be handled with care during the design of a MATLAB/Simulink model as the readability of the model suffers. A similar problem is encountered when handling similar functionality at different locations in a model.

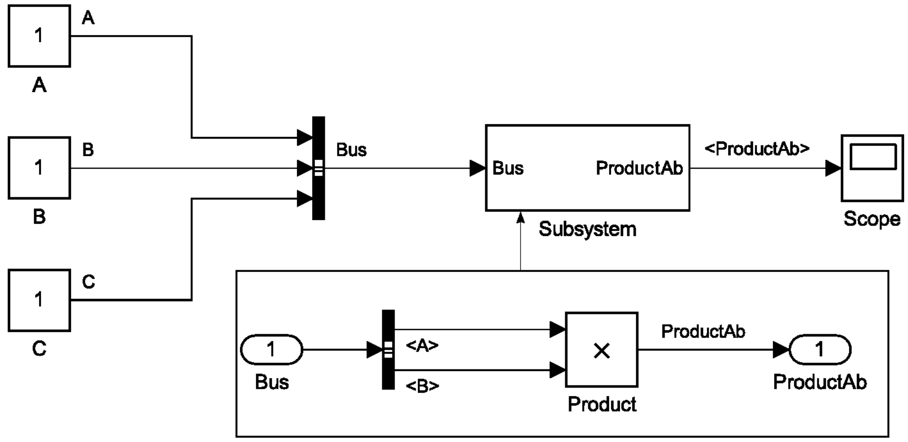


Figure 7.1: Example for the model smell *Subsystem Interface Incongruence* (see Def. 7.7)

Definition 7.6 (Duplicate Model Part)**.**
A set of model elements from a model $\mathcal{M}$ is called a duplicate model part, if there exist another set of model elements from $\mathcal{M}$ that represent the same structural and syntactical properties.

The creation and maintenance of duplicate model parts already has been identified as a potential problem in Chapter 6 and is further mentioned by Fowler as a smell occurring in traditional software development in the form of *duplicate code* [54].

7.2.3 Interface Definition

The readability and understandability of a Subsystem block not only stems from its name, but also from the interface it exposes, which is composed of the ports of the Subsystem block and their respective names. The interface of a Subsystem block should only receive signals that are actually used by the blocks it contains, especially if bus signals are part of the interface to abide to the principle of information hiding [114]. If this is not the case, the parameters of the function realized by the Subsystem block are not explicit and the interface of the subsystem is incongruent to its effectively used interface.

Definition 7.7 (Subsystem Interface Incongruence)**.**
Let $B_{nonvirtual}$ be the set of nonvirtual blocks in a MATLAB/Simulink model $\mathcal{M}$. The interface of a Subsystem block b is said to be incongruent, if b receives more signals than it actually uses.

$$C_{in} = \bigcup_{p \in ports(b)} segs_P(p)$$
$$\exists\, c \in C_{in} : \forall\, p_{dest} \in dests_S(signal_c(c)) : f_p(p_{dest}) \notin cssys_{rec}(b) \vee$$
$$(f_p(p_{dest}) \in cssys_{rec}(b) \wedge f_p(p_{dest}) \notin B_{nonvirtual})$$

Consider Figure 7.1 for an example of the aforementioned model smell. The signals A, B and C enter the displayed Subsystem block as part of the bus signal Bus but only the signals A and B are used, while signal C is terminated at the BusSelector block. Therefore, signal C is not used and the interface of the Subsystem block is incongruent to its definition. Enforcing subsystem interfaces to only contain the signals it effectively uses further increases the reusability of the subsystem, as the interface does not reflect the model environment the subsystem was initially developed in, for example by including a context specific bus as part of its interface.

One of the design decisions regarding Subsystem interface definitions is the application of a consistent definition pattern across all Subsystem interfaces of a model.

Definition 7.8 (Inconsistent Interface Definition)**.**
The interface definition of a Subsystem is inconsistent, if a subsystem of similar size exists, whose interface definition differs in overall usage of bus and atomic signals.

In particular, the input interface of subsystems of similar size and complexity might consist of bus signals in one location and of atomic signals in another location.

An interface definition further orders received and emitted signals. It is advisable, that closely related signals, e.g. a subsystem receiving two signals representing x and y coordinates within a coordinate system, should be placed in proximity to each other to highlight the semantic coherence of these signals. The actual order of signals in the interface of a Subsystem is defined by the order of its containing In-/Outport blocks.

Definition 7.9 (Non-optimal Port Order)**.**
The order of ports is said to be non-optimal, if semantically coherent signals are not located next to each other.

The size of an interface definition is another concern affecting the readability of a model. If the interface definition exceeds a certain threshold, it becomes harder for a developer to grasp an overview of the interface.

Definition 7.10 (Long Port List)**.**
A subsystem containing more than T_{Ports} In/Outport blocks is considered to possess the Long Port List model smell, with T_{Ports} being the maximum amount of acceptable In-/Outport blocks.

This smell is similar to the code smell *Long Parameter List* introduced by Fowler [54].

7.2.4 Signal Flow

The signal flow within a MATLAB/Simulink model defines data dependency relationships between the blocks in the model, as introduced in Chapter 5. Therefore, the routing of the signal flow should follow basic principles to be comprehensible and not be overly complicated. Typically, data flow is modeled from left to right on each hierarchy level, except when data is feedback, e.g. by the use of Delay blocks. This results in connected blocks to be positioned from left to right with regard to the data flow of these blocks.

One example of an overly complicate signal flow is captured by the *Redundant Signal Paths* smell, which describes the case of a subsystem receiving the same signal more than once: Either via a distinct port or encapsulated in a bus signal.

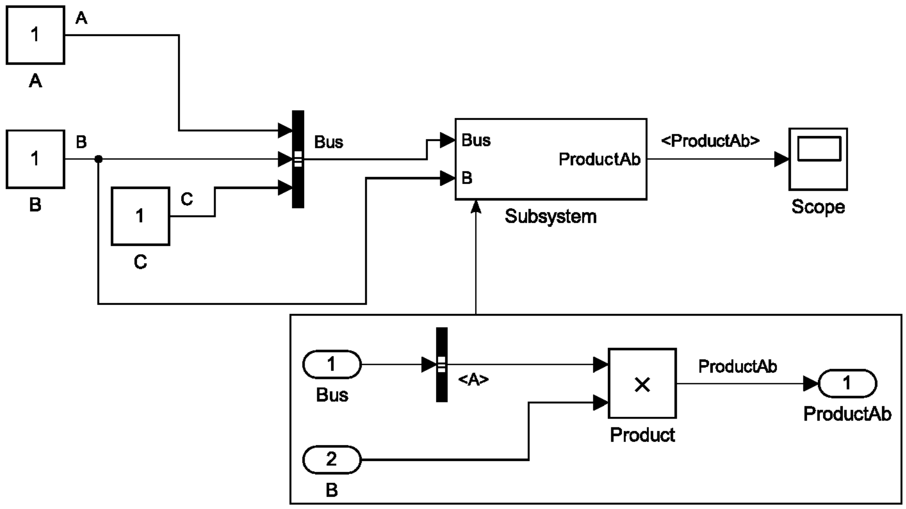


Figure 7.2: Example for the model smell *Redundant Signal Paths* (see Def. 7.11)

Definition 7.11 (Redundant Signal Paths).
A Subsystem block b is said to contain the Redundant Signal Paths smell if it holds that

$$\exists\ p_1, p_2 \in ports_{in}(b) : \Big(\bigcup_{c_1 \in segs_P(p_1)} signal(c_1) \Big) \cap \Big(\bigcup_{c_2 \in segs_P(p_2)} signal(c_2) \Big) \neq \emptyset$$

If the same signal enters a subsystem twice, possibly even with different signal name aliases, the redundant signal flow increases the visual complexity of the model and may obfuscate the source of a signal. Consider Figure 7.2 for an example of the *Redundant Signal Path* model smell. Signal B enters the displayed Subsystem block twice, once contained in the bus signal Bus and through a distinct inport. The signal instance routed through the bus signal terminates at the BusSelector contained in the Subsystem block, while the atomic signal B routed through the second inport is used as part of the computation. This overly complicates the visual complexity of the model and may hinder model maintenance in the future.

Another desired property of signal flow is that signals should only be routed into subsystems that require these signals to compute further values. If, for instance, a signal is emitted by a Subsystem block b and is again only consumed by b without any changes, the subsystem emits a *Cyclic Signal Path*.

Definition 7.12 (Cyclic Signal Path).
If a signal is emitted by a Subsystem block b that is again only consumed by b without being changed, b is said to be possessed by the Cyclic Signal Path smell. A signal is considered unchanged, even if it is propagated over a bus-capable block.

A cyclic signal path again contributes to the visual complexity of the model and could be resolved by moving the feedback loop into its emitting subsystem.

A special case of the *Cyclic Signal Path* smell occurs if a signal is feedback into its emitting subsystem and used in the same step to calculate further output signals. This breaks the principle of visually modeling the signal flow of a model from left to right.

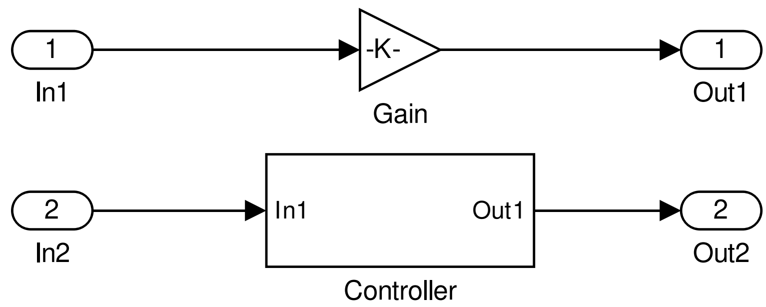


Figure 7.3: Example for the model smell *Independent Local Signal Paths* (see Def. 7.14)

Definition 7.13 (Mismatch Between Visual and Effective Flow)**.**
A mismatch between visual and effective flow occurs if a signal emitted by a Subsystem block b is feedback into b and used to compute further output signals of b in the same time step.

While the described smell could potentially occur on the same hierarchy level, we do not consider this case, as the signal flow between elements of the same hierarchy level can be visually traced, which is difficult for a signal propagated between one or multiple hierarchy levels.

As a Subsystem block is typically perceived as a unit realizing a particular function, it is expected that the input signals are used in a cohesive manner to calculate the outputs of a Subsystem block. If this is not the case, a subsystem might be affected by the *Independent Local Signal Paths* smell.

Definition 7.14 (Independent Local Signal Paths)**.**
A Subsystem block b is affected by the Independent Local Signal Paths *smell, if the directed graph G formed by the blocks and lines of b is not connected and there exist at least 2 connected components c_1, c_2 which contain at least one vertex representing an Outport block.*

Following this definition, the independent local signal paths smell represent a minimum of two independent signal flows within a Subsystem block b that do not influence each other and are both emitted via an outport of b as shown in the example model in Figure 7.3.

As signals in MATLAB/Simulink are similar to variables within a program, they should be used somewhere in the model to be relevant towards the models behavior. We consider a signal to be *used* if it terminates at a nonvirtual block. If this is not the case, the lines and virtual blocks associated with the signal do not contribute to the semantics of the model.

Definition 7.15 (Unused Signal)**.**
Let $B_{virtual}$ be the set of virtual blocks of a MATLAB/Simulink model $\mathcal{M}$. A signal s

emitted from a nonvirtual block b within a Simulink model is said to be unused in the model, if it holds that

$$\forall\ p \in dest_s(s) : parent(p) \in B_{virtual}$$

Unused signals are also relevant in the context of bus signals, i.e. if a signal is not used after it enters a bus signal via a BusCreator block, it unnecessarily bloats its containing bus signal and reduces the readability of the bus signal specification. As such a signal might still be used before it entered the bus signal, the *Unused Signal* smell does not cover this case.

Definition 7.16 (Unused Signal in Bus Signal)**.**
A signal s that enters a bus signal s_{Bus} at a block b and an outport p_{out_B} is said to be unused within s_{Bus}, if it holds that

$$\begin{aligned} \forall\ p = (L_s, (b_{dst}, p_{dst}), C) \in P_{s'} : b_{dst} \textit{ is a virtual block} \\ \textit{with } P_{s'} = \{p = (L_s, L_d, C) | \exists\ c \in C : c = c_{Bus}\} \\ \textit{and } c_{Bus} = (L_s = (b, p_{out_b}), L_D, l)\textit{: First signal segment of } s \textit{ in } s_{Bus} \end{aligned}$$

The definition captures the case that all signal paths that contain the signal segment c_{Bus} terminate at a virtual block. As the segment c_{Bus} is the first segment of s that is contained in s_{Bus}, the rule determines if all signal paths containing c_{Bus} end at a virtual block. This explicitly excludes all signal paths that do not enter the bus signal s_{Bus} at location $L_S = (b, p_{out_b})$ and therefore all signal paths that branch before reaching block b.

Another smell similar to the *Subsystem Interface Incongruence* smell occurs in a subsystem if a signal is routed through a subsystem without actually being used.

Definition 7.17 (Pass-through Signal)**.**
A signal s that enters a Subsystem block b via a signal segment c_s is said to be a pass-through signal if it holds that

$$\begin{aligned} \forall\ p = (L_s, (b_{dst}, p_{dst}), C) in P_{S'} : b_{dst} \notin cssys_{rec}(b) \\ \textit{with } P_{s'} = \{p = (L_s, L_d, C) | \exists\ c \in C : c = c_s\} \end{aligned}$$

The definition checks if all signal paths $p \in P_{s'}$ containing the signal segment c_s used to enter a subsystem b terminate at a block that is not contained in subsystem b. Consequently, signal s entering b through c_s is not used in b and can be classified as a pass-through signal. This definition does not cover multiple distinct paths of s entering b, where one is being used and one passing-through, as using such a construct is also considered to be a smell according to Definition 7.11.

Signals do not necessarily need to enter a subsystem through its ports, but can also be injected/emitted into/from a subsystem by the use of Goto/From blocks via hidden signal flow. As the use of the hidden signal flow is not reflected in the interface definition of a Subsystem block, the overall readability and reusability in another context is negatively influenced.

Definition 7.18 (Hidden Signal Flow).
Let B_{Goto} and B_{From} be the sets of Goto and From blocks of a MATLAB/Simulink model $\mathcal{M}$ and C be the set of signal segments in $\mathcal{M}$. A subsystem b_s is affected by the Hidden Signal Flow smell if either of the following conditions holds:

- $\exists\ b \in cssys_{dir}(b_S) : b \in B_{Goto} \wedge \neg\exists\ c = (L_S = (b, p_{src}), L_D = (b_{dest}, p_{dest}), l) \in C : (b_{dest} \in B_{From} \wedge \neg parent(b_{dest}) = b_s)$
- $\exists\ b \in cssys_{dir}(b_S) : b \in B_{From} \wedge \neg\exists\ c = (L_S = (b_{src}, p_{src}), L_D = (b, p_{dest}), l) \in C : (b_{src} \in B_{Goto} \wedge \neg parent(b_{src}) = b_s)$

The definitions distinguishes between the cases of a hidden signal flow being either received or emitted by a subsystem. Both cases allow the usage of Goto/From blocks in the same subsystem, so that Goto/From blocks still can be used to reduce visual complexity on one hierarchy level of the model.

7.2.5 Signal Structure

The last category targets the structure of bus signals of a MATLAB/Simulink model. As the internal structure of bus signals is hidden within the visual representation of a model, adverse design decisions can only be detected if the structure of a bus signal is explicitly examined. To help a modeler finding potential adverse design decisions in the structure of bus signals, this section introduces four smells for this purpose.

Bus signals should be used to reduce the visual complexity of a model by composing multiple lines into one line, while at the same time preserving the information transported over the composed lines. The complexity of the model is increased, if a bus contains only a single signal.

Definition 7.19 (Superfluous Bus Signal).
Let C_{Bus} be the set of signal segments contained in a bus signal s_{Bus} and C be the set of all signal segments of a MATLAB/Simulink model $\mathcal{M}$. The bus signal s_{Bus} is said to be superfluous if it holds that:

$$\forall\ c \in segs(s_{Bus}) : \neg\exists\ c_1, c_2 : c_1 \neq c_2 \wedge c_1 \subset_c c \wedge c_2 \subset_c c$$

The definition ensures that the bus signal s_{Bus} never contains more than one instance of a signal at any time, by checking if each signal segment of the bus signal at most contains one signal segment. Bus signals that do not contain any signals cannot be constructed using the Simulink UI.

As the order of ports matter in Definition 7.9: *Non-optimal Port Order*, the order of signals in a bus signals might also depends on the semantic coherence of these signals.

Definition 7.20 (Non-optimal Signal Grouping).
The signals contained in a bus signal s_{Bus} are said to be in a non-optimal order, if semantic coherent signals are not located near each other.

An example for an occurrence of this smell would be if the longitude and latitude signals are bundled into a bus signal with the name *GPS*, while the altitude signal that is also emitted by the GPS sensor, is contained in a bus signal related to the signals of an inertial measurement unit (IMU) with both bus signals being composed into another bus signal.

The only way to locally identify a signal once it has entered a bus, is by its name. If no name has been assigned to a signal entering a bus, Simulink generates a generic name for the signal, in the form *Signal X* with X being the port number of the port where the signal entered the bus signal at the corresponding BusCreator block.

Definition 7.21 (Unnamed Signal Entering Bus)**.**
A bus creator block b is said to possess the Unnamed Signal Entering Bus smell, if the FQSN of one or multiple signal segments ending at the inports of b is empty.

Potentially, a signal may be renamed on every of its signal segments. Therefore, it might happen that the same signal enters a bus twice, as a modeler could not distinguish the different instances of the signal by its name. Duplicate signals in a bus artificially bloat the complexity of the signal flow, as signals are redundantly transported through the model.

Definition 7.22 (Duplicate Signal in Bus)**.**
A bus signal s_{Bus} contains duplicates of a signal if it holds that:

$$\exists\ c \in segs_S(s_{Bus}) : |segs_{Bus}(c)| \neq |\bigcup_{c' \in segs_{Bus}(c)} signal(c')|$$

Besides the duplicate occurrences of the same signal in a bus, multiple signals might have different fully qualified signal names, but may have equal individual signal names (the last string within the sequence of the fully qualified signal name).

Definition 7.23 (Multiple Signals with same Signal Name in Bus)**.**
A bus signal s_{Bus} contains multiple signals with the same name if it holds that:

$$\begin{gathered}\exists\ c \in segs_S(s_{Bus}) : \exists\ c' \in segs_S(s_{Bus}) \wedge c' \neq c : l_n = l'_n \\ with \\ c = (L_s, L_d; \langle l_1, ...l_n \rangle) \\ c' = (L'_s, L'_d, \langle l'_1, ...l'_n \rangle)\end{gathered}$$

While signals can still be identified via their fully qualified signal name when selecting signals with the same individual name at a BusSelector block, it might be confusing for a developer, as only the individual signal names are shown in the graphical editor of MATLAB/Simulink. Consider the example model shown in Figure 7.4. Here, three bus signals are contained in bus signal *MainBus* that all contain signals with identical individual signal names. A developer trying to understand how signals were selected from the Bus, always has to check the properties of the BusSelector, as blended into the figure, as all signals have the same name and no information are available which combination of signals and in which order they were selected.

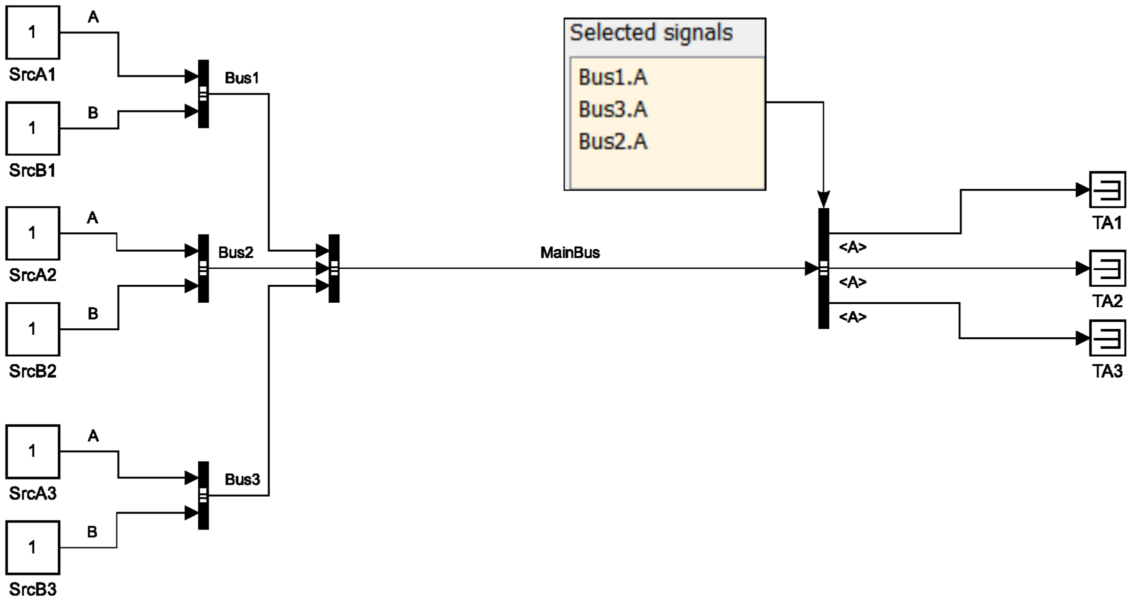


Figure 7.4: Example for the model smell *Multiple Signals with same Signal Name in Bus* (see Def. 7.23)

7.3 Detection of Model Smells

The detection of the model smells presented in the previous Section requires different analysis techniques and full access to the data of the model under review. Table 7.1 shows an overview of the techniques that can be used to detect the various model smells. Due to the definition of certain smells, these may only be manually detected using expert knowledge, such as the *Vague Name* (see Definition 7.1) smell, as the semantic meaning of the name has to be evaluated in the context of the respective model element. While the *Mismatch Between Effective & Visual Signal Flow* could potentially be found by an automatic analysis that targets signal and layout information of the model, we decided to place it in the manual inspection category, as many corner cases are involved in the visual tracing of a signal. Furthermore, depending on the layout of the model elements on a given hierarchy level, the visual flow might be arranged differently from the proposed left to right approach, e.g. top-down, top-right or even a combination of multiple possible directions meeting at certain points in the diagram. This makes it hard to design a generic algorithm for the detection of this smell.

Besides the smells that require manual inspection to be detected within a model, the rest of the smells can be automatically detected by the application of simple metrics, i.e. counting occurrences of specific elements, clone detection as introduced in Chapter 6 and signal tracing as introduced in Chapter 5. In particular, the signal graph used to capture data dependency relationships among blocks of a MATLAB/Simulink model in Chapter 5 could be leveraged to implement smell detectors for smells requiring signal

Table 7.1: Overview of analysis techniques needed for model smell detection

Method	Handled smells
Manual inspection	Vague Name Subsystem with Multiple Functions Non-Optimal Port Order Non-Optimal Signal Grouping Mismatch Between Effective & Visual Signal Flow
Metrics	Superfluous Subsystem Deeply Nested Subsystem Hierarchy Long Port List Superfluous Bus Signal
Clone detect.	Duplicate Model Part
Signal tracing	Inconsistent Interface Definition Subsystem Interface Incongruence Inconsistent Interface Definition All Signal Flow smells (see Section 7.2.4) Unnamed Signal Entering Bus Duplicate Signal in Bus Multiple Signals with same Signal Name in Bus

tracing. The definition of the smells of these categories already cover all details necessary to implement smell detectors capable of detecting these smells.

7.3.1 Implementation

While it would be possible to implement the detectors as part of the *artshop.extentions* component, we decided that, due to smells not necessarily being treated as errors, the user should be able to change how smells are detected and categorized (*warning/error*) in the framework by being able to tweak the actual detector itself. Hence, we implemented a mechanism that allows the specification of analysis rules directly in the artshop framework using the Epsilon framework [84]. The Epsilon framework provides, among others, a family of languages that can be used to validate and analyze EMF models. It defines a general-purpose language called *Epsilon Object Language* (EOL) that is an imperative programming language that can be used to create, query and modify EMF models. The feature set of the EOL includes aspects of Javascript, e.g. statement sequencing, variables, loops, import statements and if branches, and the Object Constraint Language (OCL) [158], e.g. collection querying functions. Properties and methods defined by classes of the EMF model can directly be accessed from the EOL. Moreover, native Java code can be called from EOL, which enables the reuse of already implemented framework code directly from an EOL script. User interactions are also possible via a set of pre-defined dialogs as well as the definition of user-defined functions. All further languages of the framework are derived from the basic definition of the EOL. One of these languages is the Epsilon Validation Language (EVL) that can be used to specify validation constraints on specific classes defined in the metamodel of an EMF model instance.

Epsilon Validation Language

The EVL is a validation language that allows the definition of invariants, similar to the concepts used in the OCL, i.e. invariants are evaluated on instances of classes of a given metamodel. Additionally, EVL supports the definition of guards within invariants that can prevent the evaluation of the invariant based on properties of the current instance element. An invariant can also be dependent on the result of another invariant, i.e. an invariant i_2 should only be evaluated if another invariant i_1 is satisfied, which removes the need of recurrently checking needed pre-conditions. Finally, EVL introduces the concept of constraints and critiques as refinement for an invariant. An unsatisfied constraint is interpreted as a critical error invalidating the model, while a critique indicates a non-critical situation, which should be addressed by the user. Kolovos et al. further distinguish the EVL from the OCL in Section 4.1 of the Epsilon reference document [82].

In the EVL, validation specifications are grouped in *Modules*. A validation specification can contain all constructs allowed in the EOL, i.e. user-defined operations or import statements of other Modules. In addition, a Module contains a set of contexts, which further contain sets of invariants. A Module is typically saved as a text file and can be loaded to be evaluated on an EMF model saved on the file system.

Listing 7.5 contains a validation specification for the model smell *Unnamed Signal Entering* introduced in Definition 7.21. The specification starts with the definition of a `context` that encloses a `critique` that is only applied to instances of the class specified by the context, i.e. the AbstractBlock class introduced in Section 3.3.1. The critique has a name and defines an optional guard that can be used to determine if the invariant applies to the current element instance (`self`) within a given context. In this case, the guard ensures that the critique is only evaluated on block instances representing BusCreator blocks. After the `check` label, the actual invariant of the critique is placed that must result in a boolean value. In the example, the invariant is computed by the operation `computeUnnamedSignals` that returns a set of default names for signals that were not explicitly named by the modeler and enter the BusCreator block through one of its inports. Note that an operation again has a context (second word after the `operation` keyword) and can directly be called on element instances with the type defined in the context of the operation. These names are generated as part of the signal reconstruction algorithm presented in Chapter 5. The last part of the critique defines a message that is returned if the specified invariant is not satisfied by an element.

For smells requiring a user-defined threshold, custom dialogs can be added to a module, which can be used to query these thresholds prior to the evaluation of a validation specification. As integrating these dialogs into the invariant itself would result in the dialog being shown for every evaluated element, we place a `pre` statement within a module, which is executed once before the evaluation of the specification is started. The queried value is then stored in a global variable of the scope. The `post` statement

```
/* This specification checks if a bus creator exists at which an unnamed
   signal enters a bus.
 * @author gerlitz*/
context AbstractBlock {
  constraint ms21_unnamed_signal_entering_bus {
    guard: self.type.replaceAll("\n", "") = "BusCreator"

    check : self.computeUnnamedSignals().isEmpty()

    message : "The bus creator " + self.name + "(SID: " + self.modelID + ")
     receives unnamed signals: " + self.computeUnnamedSignals().concat(", ")
  }
}

/* This operation computes the current name of a signal segment by
   considering the alias of the segment and the name of the signal.
 * @return The current name of the signal.*/
operation SignalSegment getSignalName() : String {
  var name:String;
  if(self.signalNameAlias.isDefined())
    name = self.signalNameAlias;
  else
    name = self.signal.getName();

  return name;
}

/* This operation computes the names of all signals that enter a given
   BusCreator that have no explicit name.
 * @return A set of names of unnamed signals entering the bus emited by the
     given block. */
operation AbstractBlock computeUnnamedSignals() : Set {
  var unnamedSignals : Set;
  for(port: Inport in self.inports){
    var seg:SignalSegment;
    if(not(port.virtualLines.isEmpty())){
      seg = port.virtualLines.get(0).signalSegment;
      if(seg.isDefined() and
        seg.signal.generatedName and not(seg.signalNameAlias.isDefined()))
          unnamedSignals.add(seg.getSignalName());
    }
  }
  return unnamedSignals;
}
```

Figure 7.5: EVL specification of the *Unnamed Signal Entering Bus* smell (see Def. 7.21)

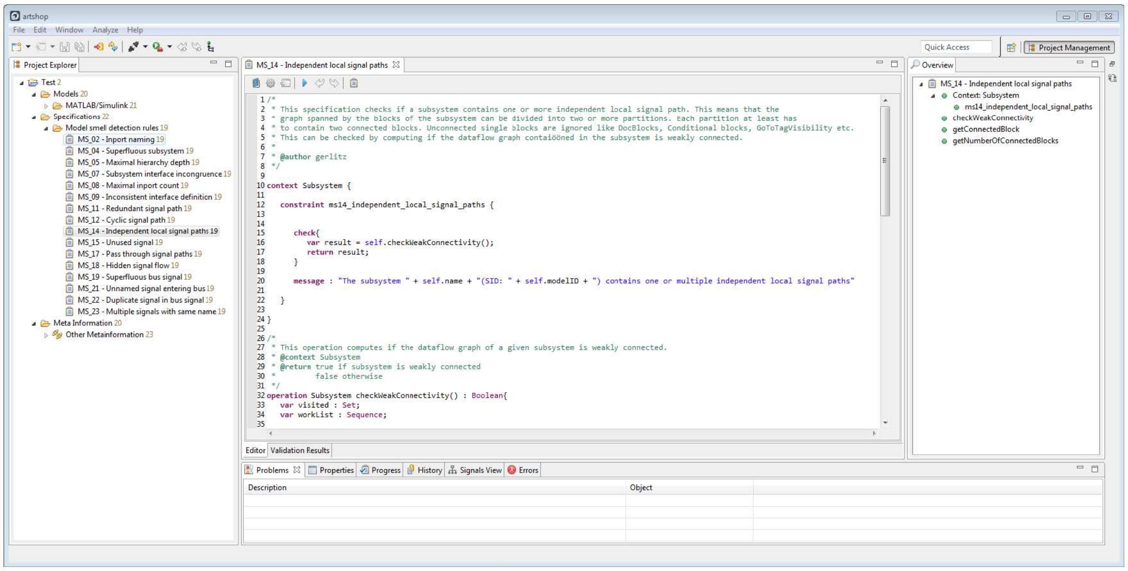


Figure 7.6: The EVL rule editor in artshop

can be used to execute EVL code after the evaluation is finished, but was not utilized during the specification of the presented model smells. Finally, each invariant can also be enhanced with a `fix` statement. This statement can be used to provide a quick fix operation specified in the EVL, to fix a problem detected by an invariant automatically. Refactoring detected model smells is not part of the scope of this chapter but a description of the general approach of fixing detected model smells is part of our joint work with Quang Minh-Tran described in [61]. The synchronous refactoring commands introduced in Section 6.3.2 could potentially also be used to realize these quick fix operations.

To evaluate a specification, both the specification and an EMF model are loaded by the framework and for each context of the validation specification, the corresponding element instances are retrieved from the EMF model and the respective invariants are evaluated on them. Unsatisfied invariants are returned by the framework for further processing. Further information about the syntax of the EVL and its internal evaluation workflow can be found in the Epsilon reference documentation [82].

7.3.2 Integration in artshop

To integrate the Epsilon framework into artshop, the framework needed to be adapted to work with models and validation specifications saved in the model repository. As the actual validation specifications are saved as plain text on the file system, a lightweight EMF model was created based on the artshop metamodel that saves the code of the EVL specification. Instances of this EMF model can then be saved alongside the models of the tool adapters in the model repository. The Epsilon framework was then extended to accept in-memory EMF model and EVL validation specification instances instead of loading them from the file system. These changes enabled evaluation of EVL validation specifications directly from the model data stored in the model repository.

A textual editor also was added to enable the specification of EVL rules in artshop. Figure 7.6 shows the EVL editor in the main view of the tool with the outline of contexts, constraints, critiques and operations being shown on the right hand side of the editor. The editor supports syntax highlighting of EVL keywords as well as comments, documentation tags (**@author, @context, ...**) and strings. Furthermore, a code completion feature was added, to support users not familiar with EVL and the actual artshop metamodels. The code completion gives hints about available EVL keywords or properties/operations available on a given model element for statement sequencing. Parser errors are directly shown within this editor.

Errors/Warnings detected by specified constraints/critiques are propagated to the artshop problem manager and shown in the problem views and all other views that support the annotation of problem tags. The messages specified in the validation specification are also included in the description of the generated tags to provide additional information about the detected problems. One example is the viewer provided by the MATLAB/Simulink tool adapter, which decorates shown blocks with tags including a tooltip containing the problem description.

7.4 Evaluation

In this section, we present the empirical evaluation results of the model smell detector presented in this chapter. The evaluation has been performed on the same system as the evaluation presented in Section 3.5. During the evaluation, we again use the models introduced in Section 3.5.1 shown in Table 3.4. To highlight the relevance of the model smells introduced in Section 7.2, we first show the amount of model smell occurrences found in the set of evaluation models. After that, we evaluate the performance of the implemented rule-based model smell detector that is used to gather these occurrences.

7.4.1 Relevance

To show the relevance of the model smells introduced in Section 7.2, we calculate the occurrences of a subset of these model smells on our set of evaluation models. During this evaluation, we exclude all model smells that require manual inspection of a model, e.g. *Vague Name* (see Def. 7.1) or the *Subsystem with Multiple Functions* smell (see Def. 7.4), as well as the *Duplicate Model Part* smell as the evaluation of the detector for this smell is already covered in Chapter 6. Table 7.2 shows the amount of detected occurrences for each considered model smell. During the evaluation of model smells that depend on a user input as the *Deeply Nested Subsystem Hierarchy* or *Long Port List* smell, we use the value 10 as the threshold referenced within the definition of the smell. It can be observed that certain smells occur very frequently as the *Unused Signal* smell, while other smells such as the *Superfluous Bus Signal* occur at a much lower rate. This is related to the fact, that certain occurrences of the *Unused Signal* smell result from the use of library blocks,

Model smell	DAS	EL	MAV	PI	ECLA
Inconsistent Port Block Name	16	14	195	591	1075
Deeply Nested Subsystem Hierarchy (T_{Level}=10)	42	0	0	0	5
Subsystem Interface Incongruence	0	1	8	39	85
Long Port List (T_{Ports}=10)	4	12	0	1	7
Redundant Signal Path	1	5	2	0	27
Cyclic Signal Path	0	0	0	0	2
Independent Local Signal Paths	7	7	2	26	17
Unused Signal	13	39	8	94	162
Unused Signal in Bus Signal	0	0	7	16	77
Pass-through Signal	3	3	9	5	17
Hidden Signal Flow	0	1	1	2	2
Superfluous Bus Signal	2	0	0	1	3
Unnamed Signal Entering Bus	0	1	1	9	3
Duplicate Signal in Bus	0	0	1	0	9
Multiple Signals with same Signal Name in Bus	0	0	0	4	12

Table 7.2: Detected model smells

which might include a superset of the actually needed functionality at a certain location within the model. Signals not used at a certain location are therefore discarded, resulting in an unused signal that does not represent a particular flaw in the model. Occurrences of the smell *Inconsistent Port Block Name*, were sometimes related to slight spelling differences between the name of the port and the signal that is propagated over it, e.g. the signal 'RindicatorLight' is propagated over a port with the name 'RIndicatorlight'. These occurrences could easily be fixed to increase the overall quality of the model. In general, it can be noted that the amount of smells found in a model, increases in relation to its size, which can be expected to a certain degree.

While inspecting individual occurrences of model smells found in the evaluation models, we also identified occurrences representing false positives resulting from the application of design patterns within the model. If for example the INMAP design pattern, which was proposed by Rau [114], is used in a model, a Subsystem block b_p is created that precedes another Subsystem block b that receives a bus signal as part of its input signals and only propagates the signal values that are actually needed by the preceded subsystem. This shifts an occurrence of the *Subsystem Interface Incongruence* smell from b to b_p although b_p was created with exactly this purpose in mind. Another example for a false positive is the occurrence of the *Redundant Signal Path* smell for a subsystem receiving a signal multiple times because it was split right before the subsystem. Due to different sources of false positives, the detector itself should be extended by a preprocessor that is able to filter occurrences due to the use of design patterns or design rationales annotated to individual elements.

Model smell	DAS	EL	MAV	PI	ECLA
Inconsistent Port Block Name	229.6	346.2	264.6	3527.4	4452.4
Deeply Nested Subsystem Hierarchy	210	184.2	99.6	900.2	1073
Subsystem Interface Incongruence	199.6	218.2	351.6	1207.6	6070
Long Port List	83.6	105.6	78.8	407.8	941
Redundant signal path	114.4	147.6	147	821	4114.6
Cyclic Signal Path	90.2	123.2	87.8	477	864
Independent Local Signal Paths	267.2	453.2	263.8	2112.8	3375.8
Unused Signal	221.4	329	274	3162.2	5440.2
Unused Signal in Bus Signal	142.2	239.8	233.6	1568.2	4681.2
Pass-through Signal	186.8	220.4	310.8	810.4	5859.4
Hidden Signal Flow	99.4	129	88.4	485.2	926.8
Superfluous Bus Signal	103.2	158	100.8	1754	2492.4
Unnamed Signal Entering Bus	108.4	185.2	117.4	2005.4	2273.2
Duplicate Signal in Bus	109.6	168.2	117.8	1776.6	2906
Multiple Signals with same Signal Name in Bus	129.6	256	187	1674.6	11009
Total	2294.6	3266.8	2723	22690.4	58701.4

Table 7.3: Average computation time (in ms) for the findings shown in Table 7.2

As an occurrence for each model smell has been found, the evaluated model smells not only represent academic assumptions but are actually instantiated in real-world models. While we retrospectively analyzed the evaluation models after they have been created, model smells should at best be detected in an incremental fashion during the development process, so that minor flaws of the design can be fixed when they are created and are not propagated through the model.

7.4.2 Performance

We also evaluated the performance of the detector based on the Epsilon framework introduced in Section 7.3. Table 7.3 shows the results of the performance evaluation. Each cell within this table shows the average computation time needed to determine the occurrences of each model smell shown in Table 7.2. These times do not include the time it takes to load the models from the model repository as they were already locally available during the evaluation. As for the occurrences of the individual smells, it is not surprising that with increasing model size, the average computation time of the model smells also increases. Again, not only the amount of blocks is influencing the performance of the detector, but also the amount of signals present in the model, as already discussed during the evaluation of the flow-based slicing algorithm in Section 5.5.1.

The resulting times indicate that model smell detection can be conducted in reasonable amount of time even for bigger models when using the Epsilon framework that first has

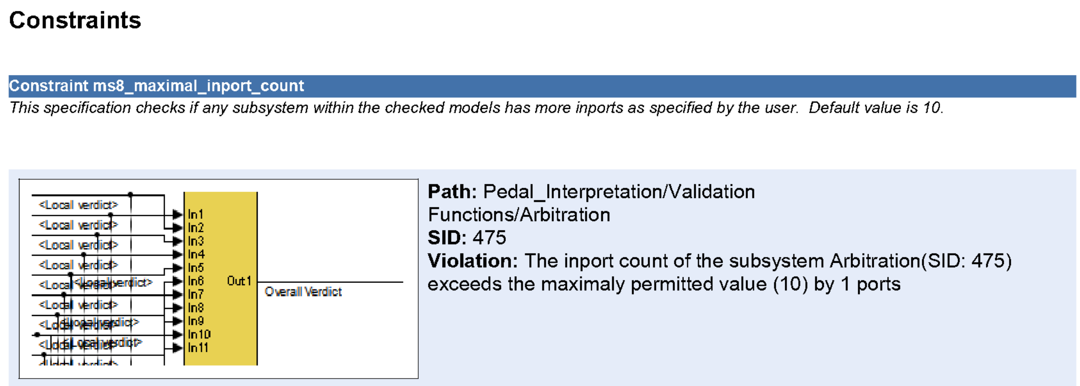


Figure 7.7: Example report for the single occurrence of the *Superfluous Bus Signal* smell in the PI model

to interpret the model smell detection rules specified with the EVL. Even for the ECLA model, all model smell occurrences could be found within a minute while detection for the PI model took 22 seconds and less than 4 seconds for the other models. To test the scalability of the approach for large-scale models, we also applied the detection rules on the version of the MAV model that has been duplicated a hundred times used for the evaluation of the MATLAB/Simulink tool adapter in Section 3.5.1. The detector took approximately 5 times longer than the hundredfold increase of the total computation value for the MAV model, resulting in a computation time of around 23 minutes. While the performance of the detector did not scale proportional to the increase in model size, the performance of rules including a lot of string manipulations and comparisons degraded, with 6/14 rules being responsible for 86 % of the computation time. This provides a starting point for future optimizations of the actual EVL rules.

7.5 Conclusion and Future Work

In this chapter, we introduced the concept of model smells for MATLAB/Simulink models inspired by the notion of code smells by Fowler [54]. While not representing syntactic or semantic errors, model smells can be related to design flaws that reduce the overall quality, understandability and maintainability of a model. In total, we provided a definition for 23 different model smell of which 15 have been implemented as part of a model smell detector based on the Epsilon framework, which has been integrated into the artshop framework. Detected results can be exported as a report as an *.pdf document as shown in Figure 7.7.

Further, we also implemented an editor for the EOL scripting language that can in the future be used for the creation of user-defined analyses. In future work, we plan to extend this editor with extensive visualization capabilities to offer potential developers the opportunity to visualize their analysis results in an appropriate way. Currently only a simple table viewer is implemented.

During this chapter, we focused on the aspects related to the detection of the smells, with a number of signal related smells being based on the definitions provided in Chapter 5. In [60] we also discussed how refactoring operations can be used to automatically resolve detected model smells and proposed the use of the SLRefactor framework developed by Quang-Minh Tran. However, in the future, it would be possible to integrate automatic refactoring operations, based on the commands presented in Section 6.3.2, within EVL rules using the `fix` statement.

During the inspection of the model smell occurrences, we detected a few false positives among the occurrences of certain model smells. These resulted from specific design decisions in the model, e.g. application of design pattern or integration of library blocks. In future work, the detector should be extended to support the detection of annotations to certain model elements that provide insights to the design rationale of specific model fragments. Annotations can then be either manually created by a developer or deduced in a pre-processing step of the smell detector. It needs to be evaluated if the trade-off of maintaining another set of annotations and traceability links is worth the reduction of false positives during the model smell detection or similar analyses.

As mentioned during the performance evaluation in Section 7.4.2, the evaluation of the prototypical model smell detection rules suffer from performance degradation when applied on large-scale models. In the future, these rules can be further optimized to increase the overall performance of the approach. Moreover, a server-side evaluation of the specified rules would save the effort of loading a local copy of the model from the repository and further increase the applicability of the approach.

8 Conclusion

In this thesis, we presented an approach for the incremental integration and subsequent static analysis of model-based automotive software artifacts as part of the artshop framework. Our approach enables the extraction and integration of model data into a repository using tool adapters that convert proprietary model data into a portable model representation derived from a well-defined metamodel. Artifacts stored in the repository can be queried and checked by the implemented static model analysis techniques utilizing the data structures provided by the respective tool adapters. The analyses can directly work on the artifacts extracted by the tool adapters and may be applied during arbitrary development stages.

8.1 Summary

A prerequisite for the application of static model analyses is the thorough extraction of model data from their respective development tools and the creation of well-defined interfaces and data structures to ease the access of model data during analysis. We approached these challenges by defining a metamodel able to represent common properties of model-based software artifacts as well as specializations of this metamodel for three specific development artifacts: functionmodels from MATLAB/Simulink, formal modules from IBM Rational DOORS and feature models, family models and variability configuration descriptions from pure::systems pure::variants. For each specialization of the metamodel, a tool adapter was created that allows the import and instantiation of these models from real-world artifacts. These artifacts can subsequently be stored within a model repository offering version control and querying capabilities. The metamodel further includes concepts to express traceability links between imported model elements as well as to artifacts outside of the repository. Moreover, a synchronization mechanism has been developed to synchronize models stored in the repository to outside changes made to their source artifacts. This allows the incremental application of changes, while preserving existing traceability links.

Based on the model repository and the data structures provided by the tool adapters, analyses can be defined in a structured manner by querying data from the repository and processing the returned data. In this thesis, we have implemented four static model analysis techniques designed for different activities of the model-based development process, targeting the primary artifact type used in automotive model-based software development: MATLAB/Simulink models. We presented a slicing approach that performs a dependency analysis based on reconstructed signal information from a MATLAB/Simulink model and is particularly useful during debugging, testing or change-impact analysis. The clone and model smell detection techniques assess a model with respect to industry-standard quality criteria and may be used to both eliminate smaller quality defects as well as preventing the accumulation of technical debt. They can potentially be integrated into a continuous integration process to assess the quality of models stored within a version control system automatically. During the development of the model smell detector, we further reused information created by other analyses, exposing synergies between different analyses as another advantage offered by a uniform analysis platform. The feature derivation and consistency checker proved to be an efficient tool during the extraction of inter-artifact traceability links and their subsequent validation.

The proposed techniques have been integrated within a software tool that guides a user through the artifact integration, synchronization and model analysis techniques using a graphical user interface. This allows domain experts unfamiliar with software engineering or the use of command-line tools to apply the methods and approaches introduced throughout this thesis. Specialists can define own analyses within the client application by using the EOL and EVL scripting languages to access and manipulate the model data stored in the repository.

Furthermore, we have evaluated the performance and functionality of all techniques on a set of real-world models taken from academic and industrial case studies and show the general applicability and suitability of the approach. We further performed stress tests of all approaches and showed that they scale adequately for the application in real-world scenarios. While not explicitly demonstrated, all approaches are suitable for use on incomplete models, i.e. models missing certain parts, with dangling lines or unconnected blocks, and can therefore be applied in early development stages.

To summarize, we believe that the integration approach in combination with the static model analysis techniques are ready to be deployed in practice. Initial feedback during presentations and demonstrations to engineers from OEMs in the automotive domain are uniformly positive showing that our approach successfully addresses real problems in day-to-day model development. The model repository and provided data structure form a solid foundation for the further development of static or dynamic analysis techniques for model-based software artifacts. One example is the analysis realized by Dernehl et al. that utilizes the model data extracted by the artshop framework to implement a static value range analysis for MATLAB/Simulink models [40, 41, 42, 43, 44]. We hope that further research on model-based software artifact analysis can benefit from our work, as we plan to publish parts of the software created during this thesis as open-source software.

8.2 Future Work

We already discussed multiple possible extensions in the individual chapters dedicated to the presented analysis techniques. Within this section, we also want to give an outlook on future work for the framework as a whole.

Currently, analyses in the artshop framework run client-side, i.e. they are locally executed on a model that was fetched from the model repository. To improve the overall workflow of the tool, the server-side execution of analyses seems promising. Besides the management of the model repository and the execution of queries, the server-side of artshop is mostly inactive. By prematurely executing analyses on available model data, analysis results can be cached and instantly made available to clients without any downtime removing the need for multiple client machines to execute the same model analyses redundantly. Nevertheless, analyzing locally available data should remain possible.

Propagating model data into the repository of the artshop framework is currently a manual process. By coupling the artshop server to a version control system, it could automatically fetch model data as it becomes available. Additionally, the integration of the complete framework in a continuous integration process with automatic report generation for stakeholders of an artifact could bring the overall model-based software development process in line with techniques used in traditional software development.

While we briefly discussed an analysis using different revisions of a model to create trend analyses (see Section 5.7.1), trend analyses are also desirable for the other algorithms presented in this thesis to monitor potential quality deterioration during the evolution of a model-based software artifact. Using the slicing algorithm presented in Chapter 5, change-impact analyses can be realized using existing algorithms and visualization techniques. Such an analysis could potentially be used to increase the performance of complex algorithms by performing them in an incremental fashion.

Finally, the long-term evaluation of the tool within an industrial model-based development process could provide an interesting step to further evaluate the accessibility, scalability and adaptability of the tool on real-world artifacts and simultaneously assess the acceptance of developers and quality engineers with regard to an analysis platform such as the artshop framework.

Bibliography

[1] K. Ahsan. Trend Analysis of Car Recalls: Evidence from the US market. *International Journal of Managing Value and Supply Chains*, 4(4):1, 2013.

[2] B. Al-Batran, B. Schätz, and B. Hummel. Semantic clone detection for model-based development of embedded systems. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 258–272, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] M. Alalfi, J. Cordy, T. Dean, M. Stephan, and A. Stevenson. Models are Code too: Near-miss Clone Detection for Simulink Models. In *28th IEEE International Conference on Software Maintenance (ICSM)*, September 2012.

[4] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. *MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations*, pages 361–375. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[5] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt. State-Based Model Slicing: A Survey. *ACM Comput. Surv.*, Aug. 2013.

[6] T. Arendt, M. Burhenne, and G. Taentzer. Defining and Checking Model Smells: A Quality Assurance Task for Models based on the Eclipse Modeling Framework. In *9th edition of the BENEVOL workshop*, 2010.

[7] H. A. Basit and Y. Dajsuren. Handling Clone Mutations in Simulink Models with VCL. *ECEASST*, 63, 2014.

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[9] M. Bender, K. Laurin, M. Lawford, V. Pantelic, A. Korobkine, J. Ong, B. Mackenzie, M. Bialy, and S. Postma. Signature Required: Making Simulink Data Flow and Interfaces Explicit. *Science of Computer Programming*, 113, Part 1:29 – 50, 2015.

[10] K. Berg, J. Bishop, and D. Muthig. Tracing Software Product Line Variability: From Problem to Solution Space. In *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, SAICSIT '05, pages 182–191, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[11] O. Berger, S. Labbene, M. Dhar, and C. Bac. Introducing OSLC, an Open Standard for Interoperability of Open Source Development Tools. In *ICSSEA 2011*, 2011.

[12] K. Berns, B. Schürmann, and M. Trapp. *Eingebettete Systeme - Systemgrundlagen und Entwicklung eingebetteter Software*. Vieweg + Teubner, 2010.

[13] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comput. Program.*, 53(3):333–352, Dec. 2004.

[14] D. W. Binkley and K. B. Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.

[15] O. Blasius. Konsistenzsicherung von artefaktübergreifenden Variabilitätsinformationen. Bachelor's thesis, RWTH Aachen, 2015.

[16] G. Böckle, P. Knauber, K. Pohl, and K. Schmid. *Software-Produktlinien. Methoden, Einführung und Praxis*. dpunkt-Verlag, Heidelberg, 2004.

[17] C. Brendle, K.-F. Hackmack, J. Kühn, M. N. Wardeh, R. Kopp, R. Rossaint, A. Stollenwerk, S. Kowalewski, B. Misgeld, S. Leonhardt, and M. Walter. In Silico Evaluation of Gas Transfer Estimation during Extracorporeal Membrane Oxygenation . In *9th IFAC Symposium on Biological and Medical Systems*, pages 499–504. IFAC-PapersOnLine, 2015.

[18] M. Broy. Automotive Software Engineering. In *Proceedings of the 25th international conference on Software engineering*, pages 719–720. IEEE Computer Society, 2003.

[19] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. *Proceedings of the IEEE*, April 2010.

[20] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013.

[21] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[22] S. Bunzel. AUTOSAR – the Standardized Software Architecture. *Informatik-Spektrum*, 34(1):79–83, 2011.

[23] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-model Level: The Fujaba Approach. *Int. J. Softw. Tools Technol. Transf.*, 6(3):203–218, Aug. 2004.

[24] D. N. Card and R. L. Glass. *Measuring Software Design Quality.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[25] J. P. Cavano and J. A. McCall. A Framework for the Measurement of Software Quality. In *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, pages 133–139, New York, NY, USA, 1978. ACM.

[26] C. Chen, X. Yan, F. Zhu, and J. Han. gApprox: Mining Frequent Approximate Patterns from a Massive Network. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 445–450. IEEE, 2007.

[27] J. Choi, A. Trögel, and I. Stürmer. TUDOOR - Ein Java Adapter für Telelogic DOORS. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme V, Schloss Dagstuhl, Germany, 2009, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 189–194, 2009.

[28] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001.

[29] A. Cmyrev, R. Noerenberg, D. Hopp, and R. Reissing. Consistency Checking of Feature Mapping Between Requirements and Test Artefacts. In *Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment*, pages 121–132. Springer London, 2013.

[30] M. Conrad, I. Fey, M. Grochtmann, and T. Klein. Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. *Informatik-Forschung und Entwicklung*, 20(1-2):3–10, 2005.

[31] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[32] W. Cunningham. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 29–30, New York, NY, USA, 1992. ACM.

[33] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. *Generative Programming for Embedded Software: An Industrial Experience Report*, pages 156–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[34] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[35] Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* PhD thesis , Technische Universiteit Eindhoven , 2015.

[36] Y. Dajsuren, M. G. van den Brand, A. Serebrenik, and S. Roubtsov. Simulink Models Are Also Software: Modularity Assessment. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pages 99–106, 2013.

[37] M. Dalgarno, D. Beuche, et al. Variant Management. In *3rd British Computer Society Configuration Management Specialist Group Conference*, volume 1, 2007.

[38] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model Clone Detection in Practice. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 57–64, New York, NY, USA, 2010. ACM.

[39] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone Detection in Automotive Model-based Development. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 603–612, New York, NY, USA, 2008. ACM.

[40] C. Dernehl. Automatic Invariant Checking for Discrete Block Diagrams using Lyapunov Functions with Sat Modulo Theory Solvers. In *European Control Conference 2016*, pages 441–446. IEEE, 2016.

[41] C. Dernehl, N. Hansen, T. Gerlitz, and S. Kowalewski. Static Value Range Analysis for Matlab/Simulink-Models . In *INFORMATIK 2015*, pages 1649–1660. Douglas W. Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt, 2015.

[42] C. Dernehl, N. Hansen, and S. Kowalewski. Abstract Interpretation of MATLAB Code with Interval Sets . In M. H. ter Beek, S. Gnesi, and A. Knapp, editors, *"Critical Systems: Formal Methods and Automated Verification: Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings*, pages 25–38. Springer International Publishing, 2016.

[43] C. Dernehl, N. Hansen, and S. Kowalewski. Combining Abstract Interpretation with Symbolic Execution for a Static Value Range Analysis of Block Diagrams . In R. De Nicola and E. Kühn, editors, *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, pages 137–152. Springer International Publishing, 2016.

[44] C. Dernehl, J. Kühn, and S. Kowalewski. Abstract Interpretation for Block Diagrams - Two Case Studies . In *13th Workshop on Model Design, Verification and Validation*, pages 20–29. CEUR, 2016.

[45] H. Dörr and I. Stürmer. JUST SIMPLIFY Heuristische Duplikaterkennung auf Modellen. In *Informatik 2013 - Informatik angepasst an Mensch, Organisation und Umwelt*, pages 2430–2442, 2013.

[46] C. Dziobek, J. Loew, W. Przystas, and J. Weiland. Functional Variants Handling in Simulink Models. In *Proc. MathWorks Virtual Automot. Conf.*, pages 1–6, 2008.

[47] C. Dziobek, T. Ringler, and F. Wohlgemuth. Herausforderungen bei der modellbasierten Entwicklung verteilter Fahrzeugfunktionen in einer verteilten Entwicklungsorganisation. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VIII, Schloss Dagstuhl, Germany, 2012, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 1–10, 2012.

[48] A. Egyed. Instant Consistency Checking for the UML. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 381–390, New York, NY, USA, 2006. ACM.

[49] T. Farkas. *Regelbasierte Konformitätsprüfung kollaborativer Artefakte*. PhD thesis, Berlin Institute of Technology, 2011.

[50] T. Farkas and H. Röbig. Automatisierte, werkzeugübergreifende Richtlinienprüfung zur Unterstützung des Automotive-Entwicklungsprozesses. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III, Schloss Dagstuhl, Germany, 15.-18. Januar 2007, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 61–73, 2007.

[51] T. Farkas, Tibor, T. Farkas, C. Hein, and T. Ritter. Automatic Evaluation of Modelling Rules and Design Guidelines. *Second Workshop From code centric to model centric software engineering: Practices, Implications and ROI*, 2006.

[52] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[53] F. Fieber, M. Huhn, and B. Rumpe. Modellqualität als Indikator für Softwarequalität: Eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, 2008.

[54] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[55] R. B. France, J. M. Bieman, S. P. Mandalaparty, B. H. C. Cheng, and A. C. Jensen. Repository for Model Driven Development (ReMoDD). In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1471–1472, 2012.

[56] T. Gerlitz. EC5.AP1.D6.2: Finale Methodik zur Konsistenzsicherung von Entwicklungsartefakten in der Funktionsentwicklung , 2015. Project deliverable: SPES_XT (unpublished).

[57] T. Gerlitz, N. Hansen, C. Dernehl, and S. Kowalewski. artshop: A Continuous Integration and Quality Assessment Framework for Model-Based Software Artifacts

. In *12. Dagstuhl-Workshop Modelbasierte Entwicklung eingebetteter Systeme (MBEES)* , pages 13–22. fortiss Technical Report, 2016.

[58] T. Gerlitz and S. Kowalewski. Architectural Analysis of MATLAB/Simulink Models with artshop. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 307–310. IEEE, 2016.

[59] T. Gerlitz and S. Kowalewski. Flow Sensitive Slicing for MATLAB/Simulink Models. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)* , pages 81–90. IEEE, 2016.

[60] T. Gerlitz, Q. Minh Tran, and C. Dziobek. Detection and Handling of Model Smells for MATLAB/Simulink Models. In *Proceedings of the 1st International Workshop on Modelling in Automotive Software Engineering*. CEUR, 2015.

[61] T. Gerlitz, S. Schake, and S. Kowalweski. Duplikatserkennung und Refactoring in Matlab/Simulink-Modellen. In *11. Dagstuhl-Workshop Modelbasierte Entwicklung eingebetteter Systeme (MBEES)* , 2015.

[62] H. Giese, M. Meyer, and R. Wagner. A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. *Proc. 4th Int. Fujaba Days 2006*, pages 56–60, 2006.

[63] M. Gleirscher, D. Ratiu, and B. Schatz. Incremental Integration of Heterogeneous Systems Views. In *ICSEM '07. International Conference on Systems Engineering and Modeling*, March 2007.

[64] N. Gold, J. Krinke, M. Harman, and D. Binkley. Issues in Clone Classification for Dataflow Languages. *Proceedings of the 4th International Workshop on Software Clones*, 2010.

[65] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[66] N. Hansen. Statische Analyse von Matlab-Simulink-Modellen mittels Intervallen. Master's thesis, RWTH Aachen, 2014.

[67] M. Harman, M. Okunlawon, B. Sivagurunathan, and S. Danicic. Slice-Based Measurement of Coupling. In R. Harrison, editor, *19th ICSE, Workshop on Process Modelling and Empirical Studies of Software Evolution*, Boston, Massachusetts, USA, May 1997.

[68] C. Hein, T. Ritter, and M. Wagner. Model-driven Tool Integration with ModelBus. In *11th International Conference on Enterprise Information Systems (ICEIS)*, May 2009.

[69] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, Sept 1981.

[70] A. Horváth, I. Ráth, and R. Rizzi Starr. Massif - The love child of Matlab Simulink and Eclipse. `https://www.eclipsecon.org/na2015/session/massif-love-child-matlab-simulink-and-eclipse`. Accessed: 2016-01-05.

[71] W. Hu, T. Loeffler, and J. Wegener. Quality Model based on ISO/IEC 9126 for Internal Quality of MATLAB/Simulink/Stateflow Models. In *Industrial Technology (ICIT), 2012 IEEE International Conference on*, pages 325–330, March 2012.

[72] B. Hummel, E. Juergens, and D. Steidl. Index-based Model Clone Detection. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 21–27, New York, NY, USA, 2011. ACM.

[73] ISO. 26262-6 - Road vehicles - Functional safety - Part 6 Product Development Software Level, 2011.

[74] ISO/IEC. ISO/IEC 9126 - Software Engineering - Product Quality, 2001.

[75] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[76] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990.

[77] S. Kemmann, T. Kuhn, and M. Trapp. Extensible and Automated Model-Evaluations with INProVE. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6598 LNCS, 2011.

[78] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[79] L. K. Klauske. *Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus.* PhD thesis, Fakultät IV - Elektrotechnik und Informatik der Technischen Universität Berlin, October 2012.

[80] L. K. Klauske, C. D. Schulze, M. Spönemann, and R. von Hanxleden. Improved Layout for Data Flow Diagrams with Port Constraints. In *Proceedings of the 7th International Conference on Diagrammatic Representation and Inference*, Diagrams'12, pages 65–79, Berlin, Heidelberg, 2012. Springer-Verlag.

[81] C. Kolassa, D. Dieckow, M. Hirsch, U. Creutzburg, C. Siemers, and B. Rumpe. Objektorientierte Graphendarstellung von Simulink-Modellen zur einfachen Analyse und Transformation. *Tagungsband AALE 2013*, 10:277–286, 2013.

[82] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige. *The Epsilon Book*. online, 2016.

[83] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.

[84] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. *Rigorous Methods for Software Construction and Analysis*, chapter On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[85] R. Koschke. Survey of Research on Software Clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[86] M. Kowal and I. Schaefer. Incremental Consistency Checking in Delta-oriented UML-Models for Automation Systems. In *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FMSPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016.*, pages 32–45, 2016.

[87] M. Kugelmeier. Integration von IBM Rational DOORS in das artshop Modellrepository. Bachelor's thesis, RWTH Aachen, 2014.

[88] J. T. Lallchandani and R. Mall. Slicing UML Architectural Models. *SIGSOFT Softw. Eng. Notes*, 33(3):4:1–4:9, May 2008.

[89] K. Lano and S. Kolahdouz-Rahimi. *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, chapter Slicing of UML Models Using Model Transformations, pages 228–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[90] C. Legat, J. Folmer, and B. Vogel-Heuser. Evolution in Industrial Plant Automation: A Case Study. *IECON Proc. (Industrial Electron. Conf.*, pages 4386–4391, 2013.

[91] E. Legros, W. Schäfer, A. Schürr, and I. Stürmer. MATE: A Model Analysis and Transformation Environment for MATLAB Simulink. *Model. Eng. Embed. Real-Time Syst.*, 6100:323–328, 2010.

[92] A. Leitner, B. Herbst, and R. Mathijssen. *Lessons Learned from Tool Integration with OSLC*, pages 242–254. Springer International Publishing, 2016.

[93] T. J. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[94] C. Mengi and M. Nagl. Refactoring of Automotive Models to Handle the Variant Problem. *Softwaretechnik-Trends*, 32(2), 2012.

[95] D. Merschen. *Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software* . PhD thesis , Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University , March 2014.

[96] D. Merschen, Y. Duhr, T. Ringler, B. Hedenetz, and S. Kowalewski. Model-Based Analysis of Design Artefacts Applying an Annotation Concept . In *Software Engineering 2012 (SE 2012)*. Gesellschaft für Informatik e.V. (GI), March 2012.

[97] D. Merschen, R. Gleis, J. Pott, and S. Kowalewski. Analysis of Simulink Models Using Databases and Model Transformations. In R. Machado, R. Maciel, J. Rubin, and G. Botterweck, editors, *Model-Based Methodologies for Pervasive and Embedded Software*, volume 7706 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin Heidelberg, 2013.

[98] C. Meyer, P. Hartmann, and D. Moormann. Wind Tunnel Investigation of Stationary Straight-Lined Flight of Tiltwings Considering Vertical Airspeeds. In *IMAV2015: International Micro Air Vehicle Conference and Competition 2015*, 2015.

[99] T. M. Meyers and D. Binkley. An Empirical Study of Slice-based Cohesion and Coupling Metrics. *ACM Trans. Softw. Eng. Methodol.*, 17(1):2:1–2:27, Dec. 2007.

[100] A. Michailidis, U. Spieth, T. Ringler, B. Hedenetz, and S. Kowalewski. Test Front Loading in Early Stages of Automotive Software Development based on AUTOSAR. In *Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 435–440, March 2010.

[101] Microsoft Corporation. MS-OAUTH: OLE Automation Protocol.

[102] J. Nehring-Wirxel. Rekonstruktion von Matlab/Simulink Modellen aus dem artshop-Modellrepository. Bachelor's thesis, RWTH Aachen, 2015.

[103] S. Nejati, M. Sabetzadeh, D. Falessi, L. Briand, and T. Coq. A SysML-based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies. *Inf. Softw. Technol.*, 54(6), June 2012.

[104] M. Olszewska. Simulink-Specific Design Quality Metrics. Technical Report 1002, Turku Centre for Computer Science, 2011.

[105] M. Olszewska, Y. Dajsuren, H. Altinger, A. Serebrenik, M. Waldén, and M. G. J. van den Brand. Tailoring Complexity Metrics for Simulink Models. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, pages 5:1–5:7, New York, NY, USA, 2016. ACM.

[106] L. M. Ott and J. J. Thuss. Slice Based Metrics for Estimating Cohesion. In *[1993] Proceedings First International Software Metrics Symposium*, pages 71–81, May 1993.

[107] K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *SIGPLAN Not.*, 19(5):177–184, Apr. 1984.

[108] V. Pantelic, S. Postma, M. Lawford, A. Korobkine, B. Mackenzie, J. Ong, and M. Bender. A Toolset for Simulink - Improving Software Engineering Practices in Development with Simulink. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 50–61, 2015.

[109] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and Accurate Clone Detection in Graph-based Models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[110] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[111] K. Pohl, M. Broy, H. Daembkes, and H. Hönninger. *Advanced Model-Based Engineering of Embedded Systems: Extensions of the SPES 2020 Methodology*. Springer, 2016.

[112] V. Preoteasa, I. Dragomir, and S. Tripakis. Type Inference of Simulink Hierarchical Block Diagrams in Isabelle. *arXiv preprint arXiv:1612.05494*, 2016.

[113] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software Engineering for Automotive Systems: A Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[114] A. Rau. On Model-Based Development: A Pattern for Strong Interfaces in SIMULINK. *Gesellschaft für Informatik, FG*, 2(1):12, 2002.

[115] R. Reicherdt. *A Framework for the Automatic Verification of Discrete-Time MATLAB Simulink Models using Boogie*. PhD thesis , Technische Universität Berlin, July 2015.

[116] R. Reicherdt and S. Glesner. Slicing MATLAB Simulink Models. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 551–561, June 2012.

[117] R. Reicherdt and S. Glesner. Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie. In *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, Cham, 2014. Springer International Publishing.

[118] CQSE GmbH. Simulink Library for Java. https://www.cqse.eu/en/products/simulink-library-for-java/overview/. Accessed: 2016-03-11.

[119] Eclipse Foundation. Connected Data Objects. http://www.eclipse.org/cdo/documentation/. Accessed: 2016-01-26.

[120] Esterel Technologies. SCADE. http://www.esterel-technologies.com/products/scade-suite/. Accessed: 2016-08-30.

[121] ETAS GmbH. ASCET. http://www.etas.com/de/products/ascet_software_products.php. Accessed: 2016-08-30.

[122] IBM. IBM Rational DOORS - The DXL Reference Manual, 2016.

[123] Model Engineering Solutions GmbH. MES Model Examiner (MXAM). http://www.model-engineers.com/de/model-examiner.html. Accessed: 2016-03-18.

[124] Modelica Association. Modelica. https://www.modelica.org/. Accessed: 2016-08-30.

[125] National Instruments. LabView. http://www.ni.com/labview/. Accessed: 2016-08-30.

[126] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, June 2013.

[127] The MathWorks Inc. MATLAB. http://mathworks.com/products/matlab/. Accessed: 2016-08-30.

[128] The Mathworks Inc. Matlab Model Advisor. https://de.mathworks.com/help/simulink/ug/consult-the-model-advisor.html. Accessed: 2017-02-16.

[129] The MathWorks Inc. MATLAB/Simulink. http://mathworks.com/products/simulink/. Accessed: 2016-08-30.

[130] The MathWorks Inc. MATLAB/Stateflow. http://mathworks.com/products/stateflow/. Accessed: 2016-08-30.

[131] The MathWorks Inc. The MathWorks Automotive Advisory Board (MAAB: Control Algorithm Modeling Guidelines Using MATLAB, 2012.

[132] The MathWorks Inc. MATLAB/Stateflow - User Guide, 2016.

[133] R. Rousseau. Visualisierungskonzepte für Matlab/Simulink Modelle. Bachelor's thesis, RWTH Aachen, 2015.

[134] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.

[135] I. Schäfer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. *Delta-Oriented Programming of Software Product Lines*, pages 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[136] S. Schake. Duplikatserkennung und Refactoring für Matlab-Simulink Modelle. Bachelor's thesis, RWTH Aachen, 2014.

[137] B. Schätz, M. Broy, F. Huber, J. Philipps, W. Prenninger, A. Pretschner, and B. Rumpe. Model-Based Software and Systems Development. White paper, Department of Computer Science, Technical University Munich, 2004.

[138] J. Scheible. Ein Framework zur automatisierten Ermittlung der Modellqualität bei eingebetteten Systemen. 6. Dagstuhl-Workshop MBEES 2010: Modellbasierte Entwicklung eingebetteter Systeme. In *6. Dagstuhl-Workshop Modelbasierte Entwicklung eingebetteter Systeme (MBEES)* , 2010.

[139] J. Scheible. Ein Qualitätsmodell zur automatisierten Ermittlung der Modellqualität bei eingebetteten Systemen. . In *INFORMATIK 2010: Service Science: Neue Perspektiven für die Informatik*, volume P-176 of *Lecture Notes in Informatics (LNI)*, pages 509–514. Köllen Druck+Verlag GmbH, Bonn, 2010.

[140] J. Scheible. *Automatisierte Qualitätsbewertung am Beispiel von MATLAB Simulink-Modellen in der Automobil-Domäne* . PhD thesis , Mathematisch-Naturwissenschaftliche Fakultät der Universität Tübingen, July 2012.

[141] J. Scheible and H. Pohlheim. Automated Model Quality Rating of Embedded Systems. In *4. Workshop zur Software-Qualitätsmodellierung und-bewertung (SQMB 2011)*, 2011.

[142] M. Schulze, J. Mauersberger, and D. Beuche. Functional Safety and Variability: Can It Be Brought Together? In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 236–243, New York, NY, USA, 2013. ACM.

[143] B. Selic. The Pragmatics of Model-Driven Development. *IEEE software*, 20(5):19, 2003.

[144] J. Silva. A Vocabulary of Program Slicing-Based Techniques. *ACM computing surveys (CSUR)*, 44(3):12, 2012.

[145] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[146] M. Stephan and J. R. Cordy. Identification of Simulink Model Antipattern Instances Using Model Clone Detection. In *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2015*, pages 276–285, Sept 2015.

[147] P. Struss and C. Price. Model-Based Systems in the Automotive Industry. *AI magazine*, 24(4):17, 2003.

[148] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A.Schürr, and A. Zündorf. Das MATE Projekt - visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen. Dagstuhl Seminar Modellbasierte Entwicklung eingebetteter Systeme, 2007.

[149] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[150] J. L. Szwarcfiter and P. E. Lauer. Finding the Elementary Cycles of a Directed Graph in O (n+ m) per Cycle. Technical report, University of Newcastle upon Tyne, 1974.

[151] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[152] A. Toulmé. Presentation of EMF Compare Utility. *Eclipse Modeling Symposium*, 2006.

[153] Q. M. Tran and C. Dziobek. Ansatz zur Erstellung und Wartung von Simulink-Modellen durch den Einsatz von Transformationen/Refactorings und Generierungsoperationen. In *09. Dagstuhl-Workshop MBEES 2013* , pages 1–11, 2013.

[154] Q. M. Tran, B. Wilmes, and C. Dziobek. Refactoring of Simulink Diagrams via Composition of Transformation Steps. In *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, 2013.

[155] G. A. Venkatesh. The Semantic Approach to Program Slicing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 107–119, New York, NY, USA, 1991. ACM.

[156] M. Vierhauser, D. Dhungana, W. Heider, R. Rabiser, and A. Egyed. Tool Support for Incremental Consistency Checking on Variability Models. *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, pages 171–174, 2010.

[157] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider. Flexible and Scalable Consistency Checking on Product Line Variability Models. *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. - ASE '10*, page 63, 2010.

[158] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[159] D. Weir. Konformitätsanalyse und Auditing im artshop Modellrepository. Bachelor's thesis, RWTH Aachen, 2015.

[160] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[161] S. Weißleder, D. Sokenou, and B.-H. Schlingloff. Reusing State Machines for Automatic Test Generation in Product Lines. In *1st workshop on model-based testing in practice (MoTiP)*, page 19. Citeseer, 2008.

[162] N. Wiechowski, T. Gerlitz, D. Merschen, and S. Kowalewski. Ein Ansatz zum merkmalsbasierten Konsistenzmanagement in der Produktlinienentwicklung . In M. Horbach, editor, *INFORMATIK 2013 - Informatik angepasst an Mensch, Organisation und Umwelt*, volume P-220 of *Lecture Notes in Informatics (LNI)*, pages 2502–2516. Köllen Druck+Verlag GmbH, Bonn, 2013.

[163] B. Wilmes. *Hybrides Testverfahren für Simulink/TargetLink-Modelle*. PhD thesis , Fakultät IV - Elektrotechnik und Informatik der Technischen Universität Berlin, November 2014.

[164] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, Washington, DC, USA, 2002. IEEE Computer Society.

[165] J. Zander-Nowicka. *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. PhD thesis , Technische Universität Berlin, 2009.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2014-01 *	Fachgruppe Informatik: Annual Report 2014
2014-02	Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03	Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04	Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06	Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08	Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09	Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14	Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15	Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16	Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 *	Fachgruppe Informatik: Annual Report 2015
2015-02	Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05	Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06	Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus

2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic

2015-13 Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2015-14 Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines

2016-01 * Fachgruppe Informatik: Annual Report 2016

2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems

2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and JÃ¼rgen Giesl: Lower Runtime Bounds for Integer Programs

2016-04 Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution

2016-05 Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization

2016-06 Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

2016-07 Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

2016-08 Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features

2016-09 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic

2016-10 Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering

2017-01 * Fachgruppe Informatik: Annual Report 2017

2017-02 Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

2017-04 Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE

2017-05 Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems

2017-06 Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy

2017-07 Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.