Detection and Handling of Model Smells for MATLAB/Simulink Models

Thomas Gerlitz[1], Quang Minh Tran[2], and Christian Dziobek[3]

[1] Informatik 11 - Embedded Software, RWTH Aachen, Germany
gerlitz@embedded.rwth-aachen.de
[2] Daimler Center for Automotive IT Innovations (DCAITI), Berlin, Germany
quang.tranminh@dcaiti.com
[3] Daimler AG, Mercedes Benz Car Development, Sindelfingen, Germany
christian.dziobek@daimler.com

Abstract. Code smells in traditional software artifacts are common symptoms of the violation of fundamental design principles which negatively impact the quality of the resulting software product. Symptoms of code smells commonly occur in traditional software artifacts and cannot be directly mapped to model-based software artifacts. In this paper, we present a catalog for the detection and handling of *model smells* for MATLAB/Simulink, a widely used tool for model-based software development in the automotive domain. These model smells describe anti-pattern against universal quality requirements and have been collected in cooperation with an OEM from the automotive domain. To show the feasibility of detecting the proposed model smells, we realized a model smell detector and evaluated it within an industrial case study.

Keywords: MATLAB/Simulink, model-based software engineering, model refactoring, model quality, model smells

1 Introduction

Model-based software engineering (MBSE) is used extensively within the automotive domain, since it promises an increase in efficiency and quality of software development. In contrast to traditional software engineering, the developer creates graphical models, e.g. dataflow diagrams, to implement control algorithms responsible for systems deployed within a car, which are then used to generate software. While MBSE has certain advantages over traditional software engineering, similar problems occur when developing software with either development paradigm that effect the quality of the resulting software, i.e. modularity, maintainability, reusability etc. [6]. Common flaws within the design of traditional software artifacts are documented by Fowler in [4]. In this work Fowler, defines so called *code smells* whose occurrences decrease the quality of the software. For example, the code smell *Duplicated Code* describes that a software component has been implemented by duplicating already existing code. This potentially decreases the maintainability of the duplicated software component because errors have to be corrected in all duplicated code parts. Therefore, developers are compelled to avoid code smells within their software to increase its quality. Other methods to assess the quality of software include the use of software development guidelines, metrics or the use of formal methods to check specific quality criteria.

In this paper, we present a catalog of *model smells* for MATLAB/Simulink, a widely used tool for model-based software development in the automotive domain. These model smells describe anti-patterns against universal quality requirements and have been collected in cooperation with an OEM from the automotive domain. In contrast to modeling guidelines that need to be followed to create a syntactically consistent model, our proposed catalog of model smells can be used to detect and avoid flaws within the architectural design of a model that may complicate maintenance or further development.

The rest of the paper is structured as follows. Section 2 gives an introduction to the quality criterions the model smells are based on. After that, Section 3 contains an overview of needed techniques to detect and handle the model smells introduced in the previous section, in addition to an overview of related work in the field of quality assessment of MATLAB/Simulink models. The evaluation of a prototypical implementation of the presented model smells are contained in Section 4. Related work in the context of quality criteria for MATLAB/Simulink models is discussed in Section 5. The paper is concluded in Section 6.

2 Model Smells Catalog for Simulink

Based on our experience with creating and maintaining Simulink models as well as interacting with Simulink modelers in the automotive domain on a regular basis, we have observed that certain model properties need to be designed with care to achieve high quality models. In this section we present five model quality properties that influence the overall quality of a Simulink model. One of these properties is the naming scheme that is applied to elements of the model which is a specific attribute of each individual model element and is responsible for the proper documentation of the model. The other properties are of architectural nature and include the structural partitioning of the model in hindsight to modularization and abstraction, interfaces of Subsystem blocks, the signal flow within the model and the signal structure that is associated with this signal flow.

For each of these properties we present a catalog of quality anti-patterns or *model smells* as shown in Fig. 1. These model smells represent examples of quality flaws within these properties and are explained in the following paragraphs. Some of the presented smells are inspired by code smells [4] but the catalog also includes smells that are only applicable to Simulink models. How these smells should be detected and handled by the developer is explained in Section 3.

Name. Names of modeling elements, i.e. blocks, subsystems, lines and ports, of a Simulink model serve as documentation. In particular, a line name describes the signal or a group of signals (in case of a bus signal), that it carries. The name of a subsystem or a custom library block is a textual description of the function

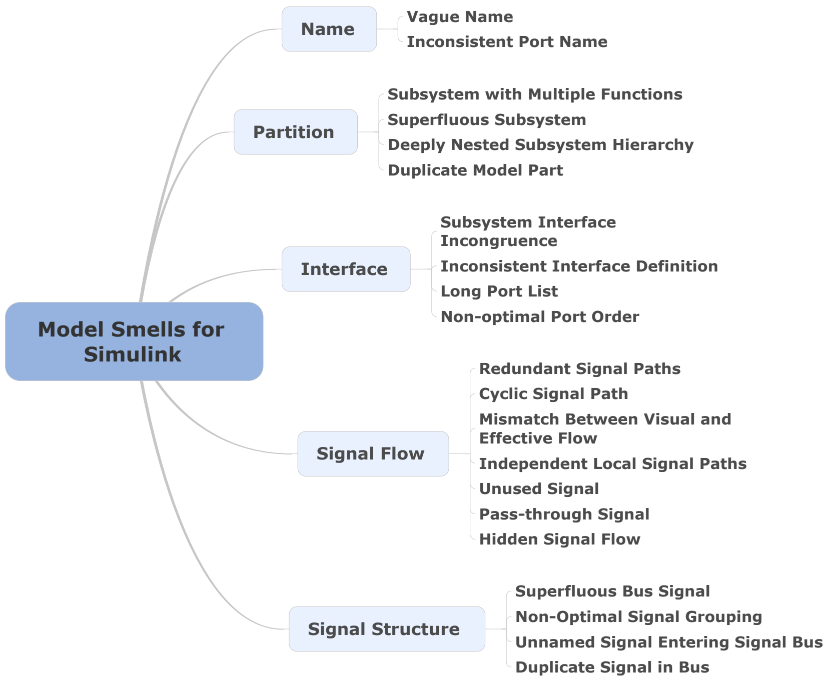

Fig. 1. Overview of the identified model smells

realized by the subsystem or the custom library block. Port names describe the input and output interface of subsystems.

The model smell *Vague Name* exists in a model if some modeling element is ill-named. For instance, given a subsystem that defines an algorithm to check if the vehicle is currently moving. A name such as *calculate_signal* would be too general. A possible name would be *check_if_vehicle_moving*. If a signal is routed across hierarchy levels via *Inport* and *Outport* blocks, these *Inport* and *Outport* blocks should be named after the signal they transport to handle the signal consistently on all levels it is propagated to. This helps to improve the readability of the model. If there is a mismatch between the signal name and the port names, the model is said to contain the model smell *Inconsistent Port Name*.

Partition. The decomposition of functions into subsystems has a huge impact on the understandability and extensibility, among others, of a model. Typically, each subsystem should implement a single well-defined function with regard to the abstraction level, on which the subsystem is located.

The model smell *Subsystem with Multiple Functions* occurs if a subsystem on the lowest hierarchy level implements multiple functions. For instance, a subsystem called *vehicle_moving* checks if the vehicle is moving. Inside the subsystem, there are two model parts. One part checks whether the vehicle velocity signal is valid or not. The other part uses the result of the former part and check if the vehicle is currently moving. It can be noticed that the subsystem contains two well-defined functions: one checks the validity of the vehicle velocity and one checks if the vehicle is moving.

A subsystem hides its content and could hamper the readability if its content is too simple or is not a well-defined function. *Superfluous Subsystem* occurs if there is a subsystem whose content is so simple that the subsystem is superfluous and thus should be resolved, for example if it only contains virtual blocks or contains an amount of blocks under a certain threshold. *Deeply Nested Subsystem Hierarchy* model smell occurs when the subsystem hierarchy has too many levels. A model with too deep hierarchy can be hard to understand since the logic is often hidden among many intermediate subsystem layers. *Duplicate Model Part* is caused by the existence of identical model parts in different places in the model, which are either created by accident or by actively copying parts of a model.

Interface. The interface of a subsystem is visually represented by its inports and outports. As a result, a subsystem with too many inports or outports can be difficult to grasp. Such a subsystem with too many inports or outports causes the model smell *Long Port List*.

The statement that the inports and outports of a subsystem define the interface of the subsystem is only valid in the absence of buses. The reason is, if no buses are used, each inport represents a single incoming signal and each outport represents a single outgoing signal of the subsystem. However, the use of buses means that a large number of signals can be transported into the subsystem via a bus through one single inport. If only a subset of those signals are used within the subsystem, the input interface becomes less obvious. We define the *potential input interface* of a subsystem as the set of all elementary signals being transported into the subsystem, both as elementary signals or in buses. Furthermore, the signals in the potential input interface, that are used within the subsystem, make up the *effective input interface* of the subsystem.

The *Subsystem Interface Incongruence* model smells occurs when the effective input interface of a subsystem is not identical to its potential input interface. The occurrence of this smell decreases the understandability of the model because it is not obvious which signals are needed by the subsystem, by looking at the incoming lines of the subsystem. An *Inconsistent Interface Definition* occurs when the definition of input or output interface of subsystems does not follow a consistent pattern. In particular, at one place, a subsystem receives signals via buses while at another place a subsystem of comparable size and complexity receives signals as elementary signals.

The order of the inports or outports of a subsystem affects the readability of the subsystem. In particular, closely related ports should be located close to each other. For instance, for a subsystem that needs both the x and y coordinate as input arguments, two inports for x and y coordinate should be placed next to each other. If a model does not follow this, it is said to have *Non-optimal Port Order* model smell.

Signal flow. This category contains model smells related to signal flow. As a general rule, the data flow in a Simulink model is from left to right, except when signals are fedback, for instance, via *Unit Delay* blocks. As a result, blocks and subsystems should be typically positioned from left to right according to data flow. Moreover, the visual flow represented by the graphical lines should reflect the real data flow happening during the simulation. This is important because the modeler tends to make assumption about the data flow based on the visual flow.

Redundant Signal Paths occurs when the same signal is routed to a subsystem via more than one path. The existence of a *Cyclic Signal Path* says that there is a signal that is an outgoing signal of a subsystem and is rerouted back to the same subsystem. If the visual flow does not reflect the data flow, *Mismatch Between Visual and data Flow* occurs.

A subsystem contains *Independent Local Signal Paths* if there exist two or more paths within a subsystem which do not influence each other. This represents a hint that the subsystem includes more than one distinct function.

The model smell *Unused Signal* exists when there is a signal that is not used. A common case of this smell is that a signal selected from a bus by a *Bus Selector* block is not used. *Pass-through Signal* is a signal that is passed through a subsystem without being used in the subsystem. *Hidden Signal Flow* occurs when the signal flow can not be recognized directly based on the graphical lines of the model. This happens when *Goto/From* and *Data Store Read/Write* blocks are used to propagate signals across system borders.

Signal Structure. The need for grouping signals into buses arises at the latest when there are too many lines crossing each other making it hard to follow the signal connections. In Simulink, the signal hierarchy in a bus corresponds to the order, in which the signals are grouped into the bus. This corresponds to the order, in which the signals are shown when the modeler wants to select a signal in the bus. As a result, the grouping of signals into buses should be well thought out.

In some cases, a bus with only a few signals, e.g. one or two, is superfluous. A model with such a bus is said to have a *Superfluous Bus Signal* model smell. *Non-Optional Signal Grouping* occurs if signals in buses are not grouped optimally. For example, functionally related signals are not close to each other in a bus. *Unnamed Signal Entering Bus* occurs when a signal without a name enters a bus. Simulink automatically assigns names *signal1*, *signal2* and so forth to unnamed signals entering a bus. When the modeler later chooses a signal from the bus, it is difficult to recognize which signals those names represent. It is also possible to bundle a signal into a sub bus of a bus B that is already contained within the bus B. This model smell is called *Duplicate signal in bus*.

3 Detection and Handling of Model Smells

To detect the model smells presented in the previous section, a myriad of different analysis techniques can be applied. Different smells may be detected us-

Method	Handled smells	Refactoring operation
Metrics	Superfluous Subsystem	Break Subsystem
	Deeply Nested Subsystem Hierarchy	Break Subsystem, Move Blocks
	Long Port List	Merge Ports, Remove Port
	Superfluous Bus Signal	Resolve Bus
Duplicate detect.	Duplicate model part	Refactor duplicate
Signal tracing	Inconsistent Interface Definition	Rename Port Names along Signal Path
	Subsystem Interface Incongruence	Create Effective Interface
	Inconsistent Interface Definition	Create Bus, Resolve Bus, Merge Ports, Split Port
	All Signal flow smells	Add Deep Signal, Remove Signal Forwards/Backwards
	Unnamed Signal Entering Bus	Rename Signal in Bus
	Duplicate Signal in Bus	Remove Signal in Bus, Add Signal to Bus
Manual inspection	Vague Name	Rename Block/Signal/Port
	Subsystem with Multiple Functions	Split subsystem, Move Blocks
	Non-optimal Port Order	Reorder Ports
	Non-optimal Signal grouping	Reorder Signals in Bus
	Mismatch between effective & visual signal flow	Depends on occurrence

Table 1. Overview of analyses needed to identify model smells

ing different levels of automation. While certain smells may be detected using automatic or semi-automatic analyses, certain smells like the *Vague name* or *Non-optimal signal grouping* smell might only be detected by manual model inspection, because of the expressiveness of the natural language. The complexity of the needed analyses also ranges from the application of simple metrics on certain model properties in the case of the *Long Port List* smell to extensive NP-hard static analyses of duplicate model structures in the *Duplicate Model Part* smell. In addition, a multitude of smells regarding the *Signal flow* model property can only be detected using signal tracing algorithms or formal methods like abstract interpretation in the case of the *Dead path* smell. After model smells have been detected by the proposed analyses, they need to be handled accordingly. In [14, 13] we have proposed a catalog of refactoring operations that can be used to automatically refactor all model smells except the *Duplicate Model Part* smell. The detection and refactoring of *Duplicate Model Part* is described in [5]. Model smells are refactored by applying a sequence of transformations to the affected model area. A detected *Inconsistent Port Name* would be refactored by changing the name of the detected port to the name of the signal that flows through it. An overview of the methods needed to detect and handle the proposed model smells is given in Table 1.

We have implemented all needed detection and refactoring methods in the tools *artshop* [5] and *SLRefactor*[13]. Nevertheless other tools/analyses can be used that can detect and in some cases handle model smells.

In [11] an automatic quality assessment framework is presented which relies on the calculation of different metrics representing quality properties within a

Simulink model. The values calculated by the metrics are aggregated within a quality model that shows the quality rating of the model with respect to a given permissible value for each quality metric. Some of the introduced metrics measure the quality of the signal architecture of the model like the amount of independent signal paths or the average size of subsystem interfaces.

Another framework which assesses and rates the quality of a given model by pre-defined or custom rules is the *INProVE* presented in [7]. While the focus of this work lies on the quality assessment process, the authors also identify a set of standard indicators expressing the quality of the model. This set also includes metrics that describe quality properties of signal flows within a model, like isolated model parts, independent signal paths and signals that enter and leave a subsystem unused (Pass-through connections).

In [2] *ConQAT*, a duplicate detection framework for MATLAB/Simulink models is presented. In the presented approach a Simulink model is first transformed into a graph with flat hierarchy and then an extended depth-first search is applied to identify isomorphic subgraphs in the model.

The metrics and analyses mentioned above can be used to identify a subset of the proposed model smells, which emphasizes the relevance of the smells addressed by this paper.

4 Evaluation

While it is technically possible to implement all model smell analysis and refactoring techniques within one tool, we will focus on the evaluation of the model smell analysis within the tool *artshop*. As mentioned in the previous section, the presented model smells can either be detected fully automatically, by automatic analysis and subsequent review of the results or by manual inspection of the model. The analyses that can be performed without manually inspecting the model, have been prototypically realized within *artshop* by the creation of user-defined conformity rules which check the occurrence of model smells within specific model elements. These rules are defined with the *Epsilon Validation language (EMF)* and are checked with the Epsilon framework against models that have been imported into the artshop model repository. The rules are checked at runtime via an interpreter and produce results that are sound as the rules check certain properties of the model that indicate a model smell.

To evaluate the performance of the prototypical implementation and show the feasibility of the implemented model smell analyses, we performed our model smell analysis on three industrial models. The tests have been conducted on a computer equipped with an Intel Core i5-3320M processor and 16 GB RAM. The model *DAS* is a model that implements the logic of a driver assistance system. The model *ELS* realizes an exterior light system, while the model *EMC* realizes the control of the exterior mirrors. The size of the analyzed models range from about 1000-1600 blocks in the model *DAS* and *ELS* to over 20000 blocks in the *EMC* model. The results of the analyses are shown in Table 2 where the number of detected model smells are displayed within each cell. The time

Model smell	**DAS**	**ELS**	**EMC**
	1043 blocks	1683 blocks	21267 blocks
Inconsistent Port Name	17	14	1296
Superfluous Subsystem	105	58	3491
Deeply Nested Subsystem Hierarchy	Variable	Variable	Variable
Subsystem Interface Incongruence	0	0	165
Long Port List	Variable	Variable	Variable
Inconsistent Interface Definition	Variable	Variable	Variable
Redundant signal path	1	5	389
Cyclic signal path	0	0	0
Independent local signal path	7	7	220
Duplicate model part	12	31	n/a
Unused signal	0	0	60
Pass-through signal	3	4	522
Hidden signal flow	22	25	58
Superfluous bus signal	2	0	67
Unnamed Signal entering bus	0	0	0
Duplicate Signal in Bus	0	0	8

Table 2. Detected model smells

needed to detect individual smells ranges from 300-400 ms within the DAS and ELS models and 3-4 s within the EMC model in addition to the time that is required to import the model into *artshop* (20-30 s for DAS/ELS and 2-3 min for EMC).

The table shows the number of detected occurrences of a subset of the presented model smell within each analyzed model. The size of the results marked as *variable* depends on an input parameter provided by the user that widens or restricts the number of results detected by these smells. It can be observed that certain smells occur frequently within the analyzed models like the *Superfluous Subsystem* smell while other smells like *Cyclic signal path* do not occur at all. In case of the *Superfluous Subsystem* smell, a multitude of this occurrences are intentional, as some subsystems only contain virtual blocks due to architectural and structural reasons, for instance due to the use of the IN-/OUTMAP pattern as described in [10]. The occurrences of the model smell *Inconsistent Port Name* sometimes were related to simple spelling differences, e.g. a port named 'RIndicatorLight' received a signal named 'RindicatorLight', but also included smells with truly different names. The fact that some smells do not occur at all shows that the model quality of the analyzed models already conform to certain quality criteria that are required by these smell categories, but does not lessen the importance of these smells. During the analysis of the results of the implemented detector, we also found false positives that were detected but do not necessarily reflect a model smell within the model, as already mentioned for the *Superfluous subsystem* smell. Another example is the occurrence of the *Redundant signal path* smell where a signal entering a subsystem is splitted just right before it enters the subsystem through two different ports. Detecting and handling false positives is planned for future work and can be realized by further restricting the scope of specific model smells on an individual basis and the attribution of model elements to announce the use of rational design patterns.

5 Related Work

The term *model smell* already has been introduced by the authors of [1] in the context of models created by the *Eclipse Modeling Framework (EMF)*, where the code smells listed in [4] are adapted to EMF models.

An established way to assess the quality of a Simulink model is the use of pre-defined model design guidelines as described in the MAAB guidelines [9]. Most of these guidelines can be automatically checked by the Matlab Model Advisor [8] that is included within MATLAB/Simulink. The MAAB guidelines includes stylistic and architectural constraints regarding the hierarchical composition and purpose of layers within a Simulink model. Constraints that focus on the interfaces of model components or signal flow as described in this paper are not included in the guidelines.

Other well known tools that can be used to check user-defined model guidelines are MESA [3] and MATE [12].

6 Conclusion

In this paper, we propose a catalog of model smells for Simulink models that serve as hints to potential structural deficits. As opposed to modeling guidelines such as MAAB guidelines, whose focus is mainly on syntactic guidelines, the model smells introduced in this paper concentrate on structural and architectural aspects of a Simulink model, e.g. signal flow, subsystem interfaces and others. Moreover, we suggest ways of eliminating the model smells by leveraging refactoring techniques for Simulink from our previous work.

For evaluation purpose, most of the model smells are formulated in the *Epsilon Validation Language* in the tool *artshop*. We applied the model smell detection rules to various industrial models and could reliably and automatically detect violated design principles. Detected model smells can then be automatically eliminated by our Simulink refactoring tool *SLRefactor*.

From numerous discussions with Simulink modelers, we are aware that the border between modeling guidelines and model smells is not always clear. We think that the relation between modeling guidelines and model smells in the context of Simulink is the same as the relation between coding guidelines and code smells in the context of textual programming. The former addresses a uniform artifact style during the development process while the latter address symptoms of structural deficits. Similar to code smells, a model smell rather is a hint than a mistake. Therefore, if the modeler can justify the existence of certain model smells, it is totally acceptable to ignore them. Currently, it is not possible to model such justifications explicitly with the Simulink modeling language.

In future work, we plan to further extend and refine the model smell catalog by structural model smells and want to discuss the relevance of further semantical model smells, like possible division by zero or possible under-/overflows. In addition, we want to suggest additional refactoring operations to handle further introduced model smells. To reduce the amount of false positives and give developers a method to justify certain design decisions, we want to introduce a set of generic attributions that can be considered by the detector to either ignore the detected smell or change the analyses in certain cases. Since we believe that automated detection and tool-supported handling of model smells can provide great benefits to increase and maintain model quality, we plan to integrate the detection and handling of model smells into a continuous integration process.

References

[1] Thorsten Arendt, Matthias Burhenne, and Gabriele 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.

[2] Florian Deissenboeck, Benjamin Hummel, and Elmar Juergens. Code clone detection in practice. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2:499–500, 2010.

[3] Tibor Farkas, Christian Hein, and Tom Ritter. Automatic evaluation of modelling rules and design guidelines. *Second Workshop "From code centric to model centric software engineering: Practices, Implications and ROI"*, 2006.

[4] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] Thomas Gerlitz, Stefan Schake, and Stefan Kowalweski. Duplikatserkennung und Refactoring in Matlab/Simulink-Modellen [Duplicate Detection and Refactoring of Matlab/Simulink models]. In *11. Dagstuhl-Workshop MBEES 2015*, pages 17–27, 2015.

[6] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2010.

[7] Sören Kemmann, Thomas Kuhn, and Mario Trapp. Extensible and automated model-evaluations with INProVE. *Lecture Notes in Computer Science*, pages 193–208, 2011.

[8] The Mathworks. Matlab model advisor.

[9] The Mathworks. MAAB: Control Algorithm Modeling Guidelines Using MATLAB, 2012.

[10] Andreas Rau. On model-based development: A pattern for strong interfaces in simulink. *Gesellschaft für Informatik, FG*, 2(1):12, 2002.

[11] Jan Scheible and Hartmut Pohlheim. Automated model quality rating of embedded systems. In *4. Workshop zur Software-Qualitätsmodellierung und-bewertung (SQMB11)*, 2011.

[12] Ingo Stürmer, Ingo Kreuz, Wilhelm Schäfer, and Andy Schürr. The mate approach: Enhanced simulink® and stateflow® model transformation. In *Proceedings of MathWorks Automotive Conference*, 2007.

[13] M. Quang Tran and Christian Dziobek. Ansatz zur Erstellung und Wartung von Simulink-Modellen durch den Einsatz von Transformationen/Refactorings und Generierungsoperationen [Approach to constructing and maintaining Simulink models based on the use of transformation/refactoring and generation operations]. In *09. Dagstuhl-Workshop MBEES 2013* , pages 1–11, 2013.

[14] Quang Minh Tran, Benjamin Wilmes, and Christian Dziobek. Refactoring of simulink diagrams via composition of transformation steps. In *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pages 140–145, 2013.