IAC-19-D5.1.2

PaTaS: Quality Assurance for Model-driven Software Development

Kilian Hoeflinger[a]*, Arno Feiden[b], Jan Sommer[a], Ayush Mani Nepal[a], Daniel Lüdtke[a]

[a] *Simulation and Software Technology, German Aerospace Center (DLR), Lilienthalplatz 7,Braunschweig 38108, Germany*, first_name.last_name@dlr.de
[b] *Directorate of Technology, Engineering and Quality, European Space Agency, Keplerlaan 1, Noordwijk NL-2200, The Netherlands*, first_name.last_name@esa.int
* Corresponding Author

Abstract

The quality of software products in safety critical applications, extensively found within the space domain, is a key success factor but also a major cost driver. To ensure high quality of the software product, quality assurance processes with quality models and metrics are applied. With these tools and processes, product assurance managers and software developers are able to quantify the quality of the software under development. Within the ESA-funded study PaTaS (Product Assurance with TASTE Study), a product quality model with software and model metrics was developed and implemented in an end-to-end model-driven software development (MDSD) life cycle demonstrator.

The goal of this study was to identify applicable concepts to maintain quality and dependability levels when MDSD is applied. This requires the definition of connected model and software quality indicators. These indicators were integrated into ESA's reference software product quality model (ECSS-Q-HB-80-04A). The resulting adapted quality model got incorporated in a model-driven software development life cycle demonstrator. To evaluate this demonstrator and the integrated quality indicators in a realistic development scenario, mission-critical parts of the command and data handling subsystem of a satellite mission were modelled and subsequently coded. The aim of the activity was to demonstrate the effect of the end-to-end life cycle in combination with the developed quality model on the final onboard software product. In this paper we present the result of the study. The focus is on the quality model for MDSD and new quality metrics for models, which can be embedded in an end-to-end model-driven product development life cycle.

Keywords: quality assurance, model metrics, quality models, model-driven development

1. Introduction

Model-Driven Software Development (MDSD) is a commonly used software development paradigm, applied in many technical domains. One of its purposes is to raise consistency within the product, by generating source code and other artefacts from various model-views. In safety critical applications, extensively found in the aerospace and automotive domains, the quality of the software product is a key success factor but also a major cost driver. To maintain a high quality software product, quality assurance processes with quality models and metrics are used to determine the quality state of the software under development. With MDSD, the quality evaluation of the product can be conducted automatically in an early phase of the development life cycle. Nevertheless, in our view the existing processes, quality models and model metrics are insufficient and not generally applicable, due to a high adaption to specific modelling technologies (Matlab Simulink, Capella, SysML, UML etc.).

Within PaTaS (Product Assurance with TASTE Study), a product quality model with software and model metrics was developed, together with an end-to-end Model-Driven Software Development Life Cycle (MDSDLC), to improve software product assurance in model-driven software product development. The goal of this study was to find applicable concepts to maintain the quality and the dependability levels, when MDSD is applied. This required the definition of interconnected model and software quality indicators. These indicators are identified and integrated with an enhanced version of European Space Agency's reference software product quality model of ECSS-Q-HB-80-04A [1] and implemented in a MDSDLC demonstrator, which is based on TASTE [2]. To evaluate this demonstrator and the integrated quality indicators, mission-critical parts of the command and data handling subsystem of a satellite mission were modelled and subsequently coded, simulating a realistic development scenario as use-case.

The rest of the paper is structured as follows. Section 2, discusses applied concepts, standards and related studies in the domain. Section 3 targets quality assurance in model-driven software development, elaborating the quality model, the model metrics and their integration in the development life cycle. In Section 4, the demonstrator design is elaborated whilst Section 5 discusses the use-case results, followed by the

conclusion in Section 6. The study was funded by the European Space Agency.

2. Background and Related Work

Model-driven development is an idea that presents both risks and opportunities for software development processes. Commercial solutions for MDSD are prevalent in safety- and mission-critical software development in the space domain (i.e. AADL, Simulink). Emerging technologies in software engineering, (i.e. Eclipse Modelling Framework) and in systems engineering (i.e. Capella, SysML2) are accelerating this trend.

MDSD changes the development approach and requires users to rethink tools and processes. Collecting and reporting quality indicators in the form of metrics is a well-established task in software development. The European Cooperation for Space Standardization (ECSS) features it as a requirement in its software product assurance standard [3] and exemplifies it in detail in the handbook [1].

Metrics do not judge quality but inform practitioners in their judgement. Collecting and evaluating metrics draws attention to software quality, potentially improving the outcome of the project (see [4] and [5]). PaTaS implements this sentiment by early and constant application as well as evaluation of the designed model metrics throughout the entire model-driven software development life cycle. Connecting the development of metrics directly to the software development process supports their validity by bearing the idea that these metrics actually represent their associated characteristics [4, 6, 7].

There have been efforts to develop quality models for MDSD in several experiments and industrial case studies (see [8] for an overview). Commercial vendors offer software solutions to evaluate basic software metrics and modelling guideline compliance for Simulink (Simulink Check [9]), AADL (AADL Inspector [10]), and UML (SDMetrics [11]). An open source initiative to evaluate metrics and “model smells” in ecore models with the EMF Refractor [12] has not made it past preliminary development phases. “Model smells” are the result of poor design and implementation and reflect missing quality attributes [13].

Typically, internal product metrics, i.e. metrics that target attributes of the source code of the software, are concerned with size, complexity, compliance to coding/modelling standards, and readability. Efficiency of the binaries or reuse rate do not target the source code itself and therefore omitted here.

Classification of these metrics varies: While Simulink Check differentiates between size and architecture metrics to measure size and complexity respectively, whereas the ECSS software product assurance handbook [1] lists size and complexity metrics under the characteristic complexity. Some scholars try to remodel classic complexity metrics to fit to models, as Halstead metrics [14] by Olszewska et al in [15] and Card and Agresti metrics [16] in [17].

MDSD offers the chance to establish metrics for modularity, an important characteristic that is mostly evaluated by hand (see [1]). Simulink Check lists markers for this characteristic under architecture, while others have made efforts to research modularity metrics for Simulink, differentiating between too much binding of modules (high coupling), and too little binding (low cohesion) [18].

3. Quality Assurance in Model-driven Software Development

3.1 Quality Model

Figure 1 displays the extended factor-criteria-metrics reference quality model, based on the reference quality model of ECSS-Q-HB-80-04A [1]. In order to effectively evaluate the quality of a product, developed by following the model-driven methodology, it is required to split the product metric into a Model Metric (MM) and a Software Metric (SWM). These quality indicators can be used to evaluate different characteristics and their sub characteristics, mapped on the product to form a quality requirement. The study focuses product quality characteristics, but the concept is also applicable for process quality characteristics.

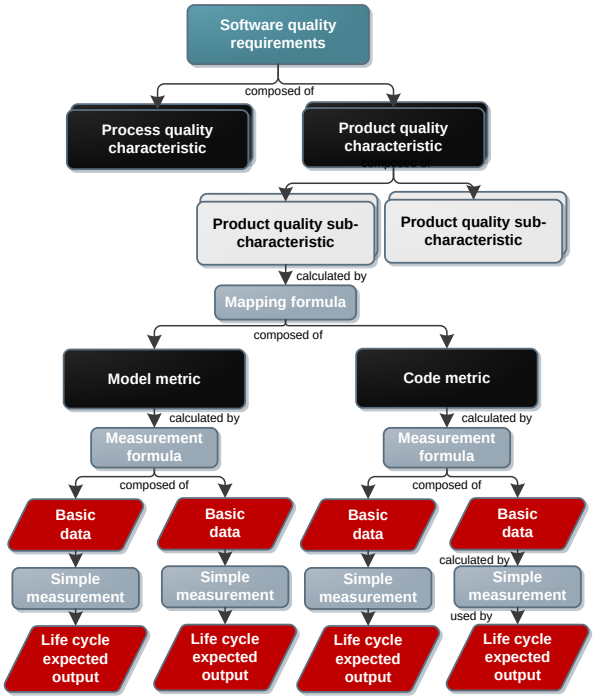

Figure 1 Extended reference quality model

To derive a verdict on the quality of the product and therefore on the fulfilment of the quality requirement, the MMs and SWMs need to be mapped against each other. This is conducted with the usage of mapping formulae. In PaTaS, three mapping formulae have been defined but further ones could be established. They are:

- *Nesting* of metrics means that a software metric is nested in a model metric. The model metric can be used to evaluate properties of the product at this rather abstract level. The result of this evaluation determines special points of interest for a subsequent metrication in software.
- *Complementation* of software and model metrics means that the state of the quality requirement of the product is depending on both metrics. Each of them have to be fulfilled to reach the desired quality.
- *Independence* of software and model metrics means that they are used only within their level of abstraction to determine the state of a quality requirement.

3.2 Model Metrics

Table 1 collects the proposed model metrics and presents the individual main and sub characteristic, which can be evaluated by them. To elaborate the functionality of the metric, its purpose and evaluation method is given, together with a threshold value. The threshold value denoted here can be used as an orientation value, as it is the result of the use-case demonstrator implementation. In general, the threshold values can vary as they are highly depending on the used model-view, modelling language and software standard (here, the Packet Utilization Standard [19] is used). For further details on how to tailor the threshold values, refer to Section 5.

To facilitate the understanding of the model metrics the following terminology shall be applied. A model type is a type specified in a modelling language.. This type represents a classification of a specific entity, i.e. a rule to create a class or a method. A model type instance is a well formed concretization of a model type, written with a modelling language, i.e. an actual class or method instantiation, defined following the semantical and syntactical model type rule [20].

Next to the classification based on their evaluable characteristics, MMs can also be grouped regarding their analytical capability. This grouping of MMs also determines a recommended order for their application and therefore the process for the resolution of exceeding MM thresholds. The identified and ordered analytic capabilities are:

1. *Conformance scanning*: Metrics with this capability force developers to create overview and standard conformance within their models. For example, model type instances or files have to be split or commented. Additionally, modelling standards shall be evaluated regarding compliance.
2. *Structural scanning*: Metrics with this capability give detailed insight on the structural design and data flow within the software product. Problematic model type instances can be identified based on the amount and kind of interconnections they have with other model type instances. Combining different metrics and targeting distinct model-views allows the investigation of various structural properties of the system.
3. *Behavioural scanning*: Metrics with this capability are related to the group of structural scanning. Nevertheless they target functional requirements and their specification. An unbalanced product specification as well as failures within the software requirements can be identified.

Table 1 PaTaS Model Metrics

Model Coupling			
Characteristic	Modularity, Balance, Complexity		
Purpose	Determining the coupling of model type instances among each other; A high coupling results in a monolithic unbalanced model/software, hindering reuse and effective maintenance, due to side effects among components.		
Evaluation	Counting references/interfaces of/to model type instances of a specific model type, used by a single model type instance. Additionally, different properties of interfaces can be used to weight them (based on Chidamber and Kemerer [21]).		
Analytical Capability	Structural scanning	**Threshold**	5..9
Model Comment Frequency			
Characteristic	Self-Descriptiveness, Complexity, Balance		

Purpose	Determining the legibility and the self-descriptiveness of the models, in order to improve the non-functional requirements.		
Evaluation	Calculating the ratio between comment lines and code lines in the model.		
Analytical Capability	Conformance scanning	**Threshold**	15..30%
Interaction Diagram Coverage			
Characteristic	Completeness, Balance		
Purpose	This MM complements the requirements implementation coverage and structural coverage SWM. A high value can indicate low functional cohesion of the model type instance, whereas, a value of zero raises questions about the general purpose of the model type instance.		
Evaluation	Counting the model type instances of a system model, used in a behavioural test model [22]		
Analytical Capability	Behavioural scanning	**Threshold**	>=1
Model Type Instances per Use-Case			
Characteristic	Modularity, Complexity, Balance, Conciseness		
Purpose	Determines the granularity of requirements and the requirements to specifications fit; A high value signifies that a change in the requirement has a great impact on the system design and implementation and it indicates a low functional cohesion, as functionality is spread over many model type instances.		
Evaluation	Counting amount of model type instances per use-case; Here, a use-case is the implementation of a test case for a software requirement (see [23]).		
Analytical Capability	Behavioural scanning	**Threshold**	5..9
Use-Cases per Model Type Instance			
Characteristic	Modularity, Complexity, Balance, Conciseness		
Purpose	This metric identifies excessively used model type instances and therefore components of the onboard software. A high value indicates that the cohesion of the model type instance might be low and that implementation failures have a broad effect on the overall system.		
Evaluation	Counting the amount of use-cases per model type instance; Here, a use-case is the implementation of a test case for a software requirement (see [23]).		
Analytical Capability	Behavioural scanning	**Threshold**	1..16
Model Type Instance Weight			
Characteristic	Complexity, Balance		
Purpose	Determines the complexity of a model type instance by counting and weighting its containing model type instances. The threshold value depends on the used indicator to determine complexity of the contained model type instances.		
Evaluation	Accumulating all model type instances, contained in a model type instance, considering a model type specific weight factor, determined by any indicator of complexity. It is the model equivalent of Weighted Methods per Class (see [21])		
Analytical Capability	Structural scanning	**Threshold**	Depends on weight factor (in PaTaS, threshold is 50..250)
Module Fan-in/out			
Characteristic	Modularity, Balance		

Purpose	High Fan-in or Fan-out indicates high complexity of the system and monolithic design, making it hard to maintain and reuse. The complexity of a procedure depends on the complexity of the control flow in the procedure and of the procedure's connection.		
Evaluation	Fan-in: Counting interfaces of local flows into a model type instance; Fan-out: Counting interfaces of local flows out of a specific model type instance; (see [24])		
Analytical Capability	Structural scanning	**Threshold**	4..6
Adherence to Modelling Conventions			
Characteristic	Modularity, Completeness, Self-Descriptiveness, Conciseness, Balance, Correctness		
Purpose	Increases maintainability as well as re-usability and is especially helpful for graphical modelling languages, as it creates overview of the system.		
Evaluation	Guidelines for the modelling, like naming conventions, consistency rules etc. Such conventions are equivalent to coding guidelines and have to be adapted to the modelling tools and domain standards. Difficult to get tool-support for the automatic evaluation. (see [25])		
Analytical Capability	Conformance scanning	**Threshold**	100%
Lines of Model Code			
Characteristic	Complexity, Balance, Self-descriptiveness		
Purpose	Indication of model complexity, balance and self-descriptiveness. Too large model files reduce the overview and therefore maintainability and re-usability. Mainly applicable for textual models.		
Evaluation	Counting the number of model lines per model file (excluding comments and blank lines)		
Analytical Capability	Conformance scanning	**Threshold**	300..500

3.3 Model-driven Software Development Life Cycle

Within this study, the MDSDLC and its phases follow the V-model development methodology. This methodology allows a good traceability and separation of modelling and coding phases. Additionally, it is frequently used at ESA and the European space community, being a standard in the development of spacecraft software. **Figure 2** visualizes the MDSDLC, which is mapped to the demonstrator design and the used tools in Section 4. During the life cycle phase execution, different modelling and QA tools are used: PaTaS domain frontend, TASTE toolchain and COTS source code analysis tools.

The PaTaS study enters the development at the Software Preliminary Design Review (SW-PDR), with predefined software system requirements and specifications in text format. It ends with the Software Critical Design Review (SW-CDR), before system verification and validation. *High Level Design* and *Detailed Design* are modelling phases. *Unit/Device Testing* and *Subsystem Verification* are coding phases. Within the *High Level Design* phase, the PUS library is designed and unit test stubs are generated for implementing them later in the *Unit/Device Testing* phase. The subsequent *Detailed Design* is used to detail the data structures and behavioural test models. From this stage, executable C++ test cases are generated for the *Subsystem Verification* phase. In all phases, either model or source code quality is determined by an analysis, which is automated as much as possible.

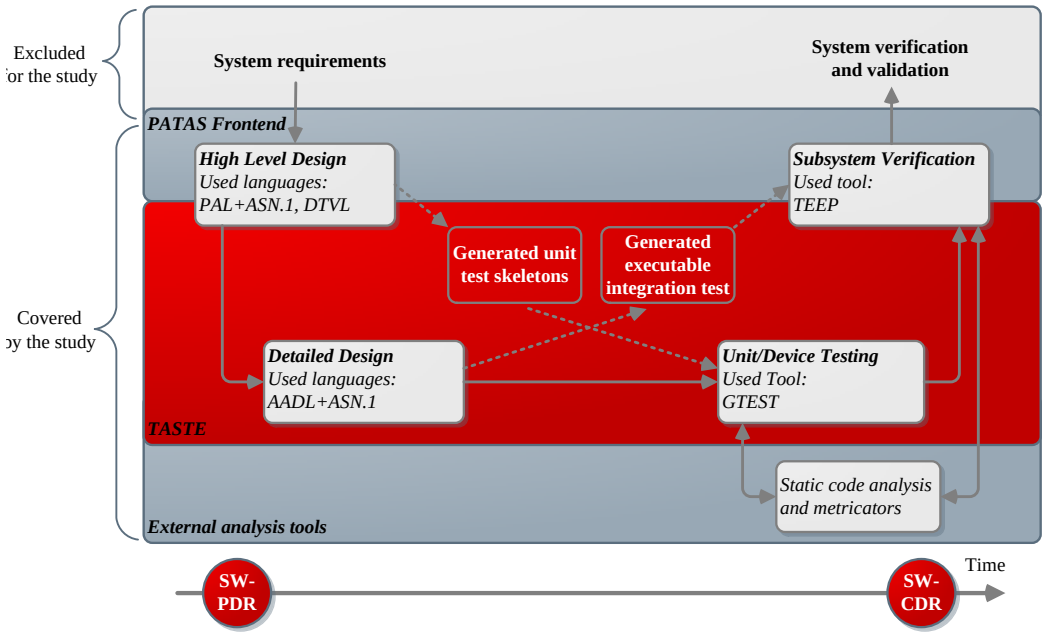

Figure 2 MDSDLC in PaTaS, based on V-model

4. Demonstrator Design and Application

This section explains the demonstrator design, focusing its application in the phases of the MDSDLC (see Figure 3). The demonstrator tool-chain is used to evaluate the MDSDLC, the quality model and the model metrics, in an end-to-end satellite onboard software use-case, based on the Space Engineering - Telemetry and Telecommand Packet Utilization Standard (PUS [19]). PUS addresses the communication between ground control and the space segment, to command or monitor platform and payload units. The standard defines extensible services, which target the base functionality of a spacecraft [1]. The demonstrator follows the Model-Driven Architecture (MDA™), adopted by the Object Management Group. MDA is a framework for the model-based development, layering the evolution from Computation Independent Models (CIM), via Platform Independent Models (PIM) to Platform Specific Models (PSM). This standard elaborates rules for the model-to-model transformation of these viewpoints [20].

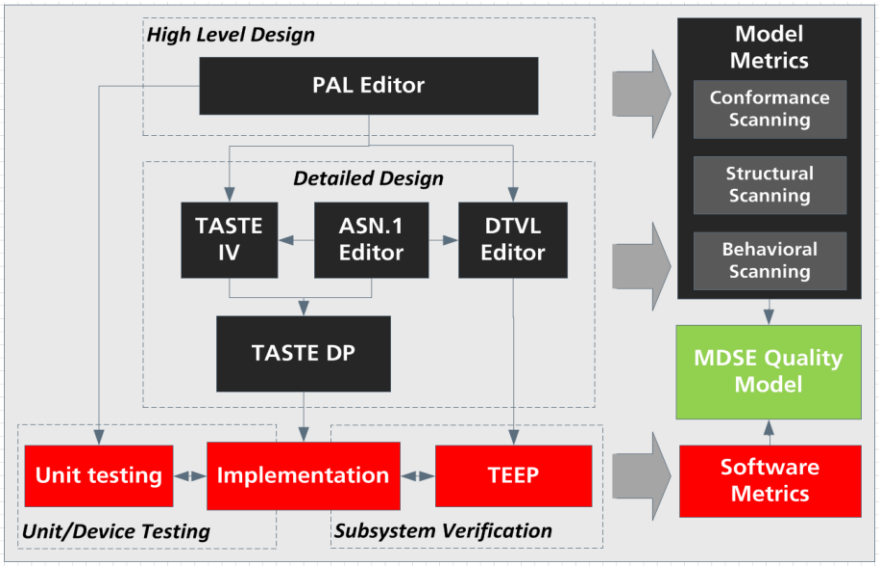

Figure 3 PaTaS demonstrator design

4.1 High Level Design Phase

Within this phase, the PUS model, consisting of applications, services and sub-types, is modelled. This structural model is described by the usage of the PUS Architectural Language (PAL) editor, a domain specific language, allowing modelling PUS-conform architectures. This editor is implemented with the Eclipse Modelling Framework (EMF) and the Xtext framework [26], as the subsequent elaborated ASN.1 and DTVL editors. Following the MDA guidelines [20], the implemented generator conducts a model-to-model transformation of the PUS architecture (CIM) to the TASTE Interface View (PIM), used in the subsequent development phase. TASTE is a model-centric software development environment and set of tools, targeting mission-critical and embedded real-time systems, developed by the European Space Agency [2, 27]. During the transformation, unit test skeletons are generated and Conformance as well as Structural scanning MMs are automatically collected. Subsequent, the MMs are investigated and the current quality model state is determined. The derived product quality verdict leads to potential action items which need to be resolved to enable a transition into the next phase.

4.2 Detailed Design Phase

In this phase, the ASN.1 editor is used to model the subtype messages, which are the parameters used in the TASTE Interface View (IV). The ASN.1 data model and the PIM of the PUS architecture are transformed into a PSM for the TASTE Deployment View. Among others, at this stage the user can configure the model for specific hardware targets, generate platform specific code skeletons and link to device drivers.

Additionally, the Data Testing and Verification Language (DTVL) editor is used to define a behavioural test model, by referencing the PUS model type instances of the PAL editor and the instantiated ASN.1 messages. The DTVL is an in-house developed editor, which is based on Linear Temporal Logic [28] and is able to describe the expected behaviour of a system over time via the TM/TC interface. It generates executable black-box test cases, which later can be executed against the onboard software. The quality is evaluated by collecting and evaluating Conformance, Structural and Behavioural Scanning MMs.

4.3 Unit/Device Testing Phase

In this phase, the transition from modelling to coding is conducted by implementing the generated source code skeletons of the PUS OBSW. The implementation of the software is conducted in a test-driven fashion with the generated unit tests. The quality of the software under development is evaluated with the help of SWM and by mapping them against MM in a product quality requirement.

4.4 Subsystem Verification Phase

This is the final phase of the PaTaS study. Here, the OBSW implementation is further tested against the executable test cases, which are described in the Detailed Design phase. These integration tests are conducted with the Test Execution and Evaluation Platform (TEEP), which is a compilable C++ test-case runtime engine, generated by the DTVL editor. This engine triggers stimuli TC messages against the OBSW and expects specific TM message(s), so called oracle messages, over time in return. Also expected periodic messages or messages that should never arrive can be defined for testing. The outputs are reported in XML and HTML5 formats as well as on the console, and indicate whether the Linear Temporal Logic property of the system holds.

5. Results and Discussion

5.1 Use-case Implementation

The implemented use-case is based on the specifications and software requirements of parts of the command and data handling of the OBSW of a SmallSat mission. The satellite has a size of about 1 cubic meter and a mass of approximately 200 kg. Within PaTaS only a small part of the OBSW was re-implemented in lab quality. Excluded were complex satellite control algorithms of subsystems, as well as driver interfaces to sensors or actuators. Table 2 denotes the use-case in figures, displaying (semi-)manually implemented components. Automatically generated source code and reports are excluded. In addition, Figure 4 shows the model in the TASTE Interface View, with the three modelled applications and their PUS onboard message dispatcher system, interfacing the EGSE. The size of the model is too large to display it in all details, due to the 90 modelled PUS subtypes interfacing the applications.

Table 2 Manually implemented part of use-case

Number of implemented TM/TC messages	90
Applications	ACS, ONS, CDH
Lines of model code	13,559
Lines of Application OBSW code	3,334
Lines of unit test code	5,845

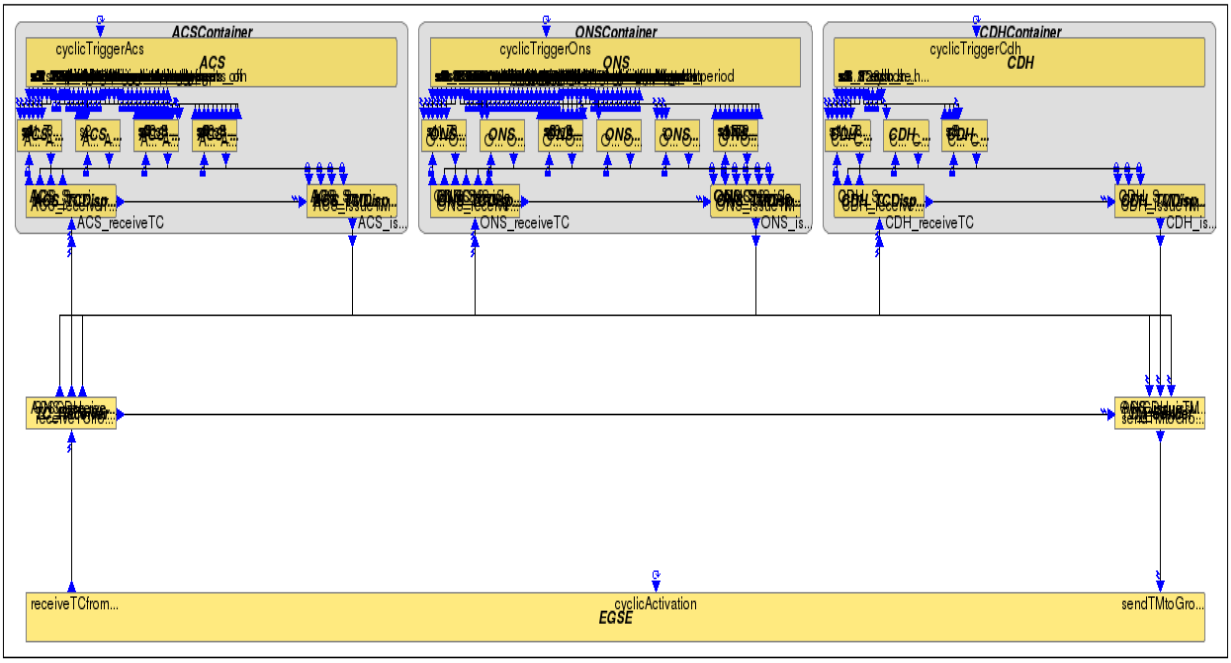

Figure 4 TAST IV model of PUS-based data handling for three onboard applications

5.2. Quality Model Application Results

Throughout the development, the instantiated quality model provides a clear overview on the current quality state as well as its progress. The use-case revealed that quality is added already in the modelling phases, and mainly has to be maintained in the coding phase. The splitting of product metrics in model and software metrics reduces the risk of a flawed design, because it allows mitigating design errors early in the development life cycle.

The mapping formulae are an important feature of the adapted quality model as it allows the introduction of custom relationships between model and software metrics as product quality requirements. This is similar to the formulae used within the metrics to determine their value from the basic measurements and could also be used for the combination of model with model or software with software metrics. Further, it is important to combine software with model metrics which are not using similar mechanism to determine a quality characteristic. The model represents the specification of the system, and is an abstraction of the source code. Measuring, for example coupling in model and source code will not reveal many new insights in the source code evaluation.

The order of evaluation and resolution of threshold exceeding metrics is important for raising the product quality. Next to the classification, based on their evaluable characteristics, MMs can be grouped regarding their analytical capability. The analytical capability determines a recommended order for the application of the MMs, and therefore the resolution of exceeding thresholds. Table 1 also denotes the *Analytic Capability* for each model metric. The identified and ordered analytic capabilities are:

- *Conformance scanning*: This group of metrics forces developers to create overview and standard conformance within their models. Model type instances or files have to be split or commented. Additionally, modelling standards shall be evaluated regarding compliance.
- *Structural scanning*: These metrics give detailed insight on the structural design and data flow within. Problematic model type instances can be identified based on the amount and kind of interconnections they have with other model type instances. Combining different metrics and targeting distinct model-views allows the investigation of various structural properties of the system.
- *Behavioural scanning*: This group of metrics is strongly related to structural scanning, but targets mainly on the functional requirement and the specification. An unbalanced system specification as well as failures within the software requirements can be identified.

Figure 5 displays the resolution of exceeding model metrics, following the aforementioned order, determined by their analytic capability.

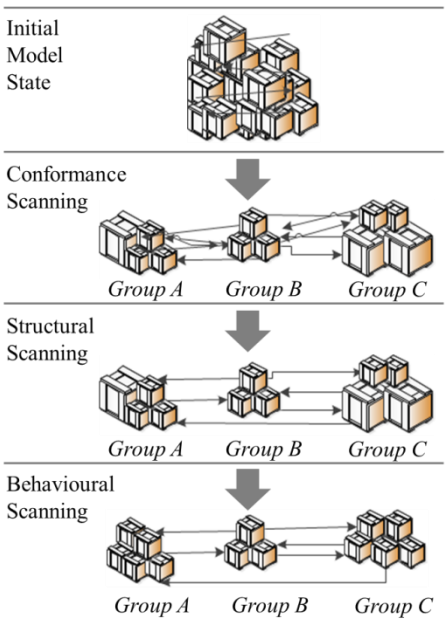

Figure 5 Application of model metrics in the MDSDLC

5.3 Model Metric Results

To reveal the expressiveness of the model metrics, two diagrams of the use-case are displayed as examples: one with the model metric *Model Coupling* and the other with the model metric *Model Type Instances per Use-Case*; For a detailed explanation of the different model metrics, please refer to

Table **1**.

Figure 6 shows the progress of the model metric *Model Coupling*. The x-axis displays time, represented as versions of the models. It measures the coupling of applications with their PUS service implementations. Here, the Onboard Navigation Subsystem (ONS) application has a very high coupling value with the PUS Function Management Service (8) in v1.0 and v1.1. As a reaction, at v1.4 this value is decreased by the introduction of a custom PUS service (here 152) removing tele-command messages from the service 8 interface. An over usage of PUS Function Management Service (8) is common in the development of onboard data handling, because the applications and their functionality grow over time. PUS Function Management Service (8) is then often used as interface from ground, instead of defining a custom service as it happened here in v1.4.

Figure 7 shows the result of the model metric *Model Type Instances per Use-Case*. Here, each software requirement, targeting the interface of the onboard data handling, is covered by a use-case. High values for messages per use-case indicate potential issues with the software requirement or the specification.

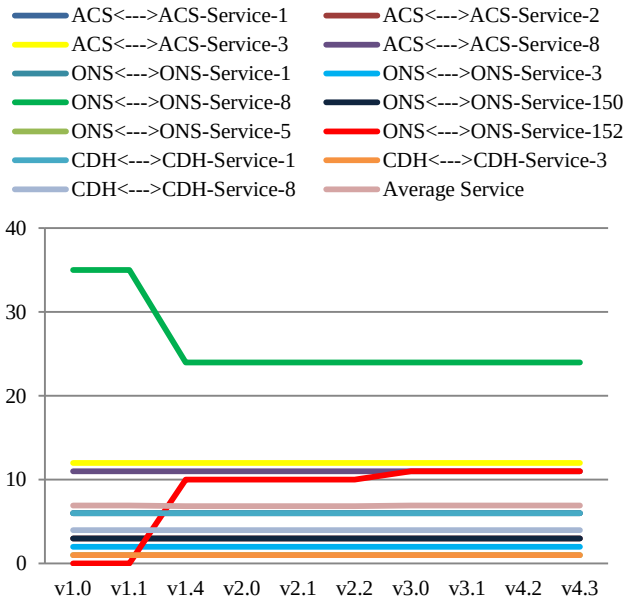

Figure 6 MM: Model Coupling

High metric values indicate that either the requirement is too coarse grained defined, meaning that there is too much functionality covered by a single requirement, or the functionality is scattered over the system by the specification. The diagram displays that over the development time, more and more use-cases are implemented (green line) and large use-cases drift towards the average use-case value. In the background, based on the value of this model metric, requirements get refined and the specification revisited to improve both.

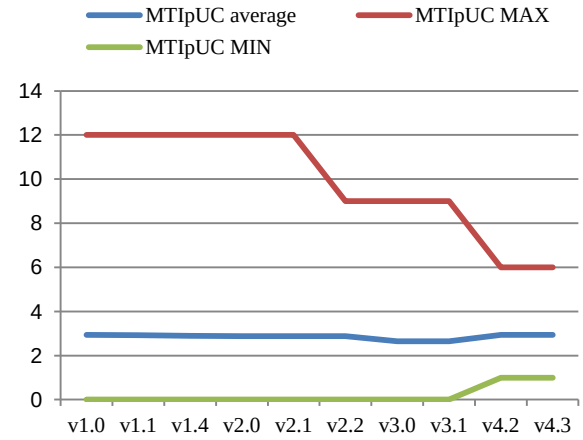

Figure 7 MM: Model Type Instances per Use-Case

Determining a threshold value for the model metrics is specifically difficult. As most of them were developed or re-designed for the evaluation of models, no experience backed by large use-cases is available. It

is recommended to maintain a small slack, being the difference between minimum and maximum measured MM value, balancing it out. This can be supported by the creation of average values.
The use-case also revealed that models and metrics evaluating them are more context sensitive than source code metrics. The models in the use-case were implementations following a domain standard (here: PUS). These standards might lead to unbalanced models with certain high values for specific model type instances. Therefore, each exceeding threshold value has to be evaluated in context of the domain standards to derive a final quality verdict.

6. Conclusions

The paper elaborates on how to raise software product quality when model-driven development is conducted, by integrating a new quality model and new model metrics into the software development life cycle. The concept is implemented in an end-to-end prototype demonstrator toolchain, based on TASTE [2, 27]. The evaluation of the demonstrator and its integrated quality indicators is conducted by the re-implementation of mission-critical parts of the command and data handling subsystem of a satellite mission, simulating a realistic development scenario as use-case.

The results reveal that the quality model, which is extended for the model-driven development methodology, and model metrics help to detect design flaws early in the development life cycle. The developed model metrics can be used to evaluate specific software product quality requirements and are combinable with software and further model metrics to derive more complex verdicts. The resolution of quality shortcomings should follow a certain order to maximise the efficiency. Additionally, the use-case showed that even for domain specific models, the product quality assessment process can be automated. A shortcoming is the determination of threshold values for the model metrics, due to the limited experience with them so far. This study can be seen as a precedency case, being a baseline for further model metric threshold investigations.

Acknowledgements

This study was funded by the European Space Agency.

References

[1] ECSS-Q-HB-80-04A – Software metrication handbook, 2011. [Online]. Available: https://ecss.nl/hbstms/ecss-q-hb-80-04a-software-metrication-handbook/.

[2] M. Perrotin, E. Conquet, J. Delange, T. Tsiodras, TASTE: An open-source tool-chain for embedded system and software development, Keplerlaan 1, 2201AG Noordwijk, The Netherlands.

[3] ECSS-Q-ST-80C Rev.1 – Software product assurance, 2017. [Online]. Available: https://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/.

[4] B. W. Boehm, J. R. Brown, M. Lipow, Quantitative Evaluation of Software Quality, in Proceedings of the 2Nd International Conference on Software Engineering, Los Alamitos, CA, USA, 1976.

[5] J.Voas, D. Kuhn, What Happened to Software Metrics?, Computer, Bd. 50, pp. 88-98, 05 2017.

[6] C. Kaner und W. Bond, Software Engineering Metrics : What Do They Measure and How Do We Know ?, Direct, Bd. 8, pp. 1-12, 2004.

[7] V. Basili, G. Caldiera, H.D. Rombach, The Goal Question Metric Approach, 1994.

[8] P. Mohagheghi, V. Dehlen, T. Neple, „Definitions and approaches to model quality in model-based software development - A review of literature, Information and Software Technology, Bd. 51, pp. 1646-1669, 10 2009.

[9] Simulink Check, Mathworks, 2019. [Online]. Available: https://www.mathworks.com/products/simulink-check.html. [Zugriff am 2019].

[10] AADL Inspector, Ellidiss Software, 2019. [Online]. Available: https://www.ellidiss.com/products/aadl-inspector/. [Zugriff am 2019].

[11] SDMetrics, SDMetrics, 2019. [Online]. Available: https://www.sdmetrics.com/. [Zugriff am 2019].

[12] EMF Refractor, Eclipse, 2019. [Online]. Available: https://www.eclipse.org/emf-refactor/. [Zugriff am 2019].

[13] H. Mumtaz, M. Alshayeb, S. Mahmood, M. Niazi, A survey on UML model smells detection techniques for software refactoring, Journal of Software: Evolution and Process, Bd. 31. e2154. 10.1002/smr.2154, 2019.

[14] M. H. Halstead, Elements of Software Science (Operating and Programming Systems Series), New York, NY, USA: Elsevier Science Inc., 1977.

[15] M. Olszewska (Plaska), Simulink-Specific Design Quality Metrics, Turku Centre for Computer Science (TUCS), Nr. 978-952-12-2564-2, 09 2011.

[16] D. N. Card,W.W. Agresti, Measuring software design complexity, Journal of Systems and Software, Bd. 8, pp. 185-197, 1988.

[17] M. Olszewska, Y. Dajsuren, H. Altinger, A.

Serebrenik, M. Waldén und J. van den Brand and Mark G., Tailoring Complexity Metrics for Simulink Models, in Proccedings of the 10th European Conference on Software Architecture Workshops, New York, NY, USA, 2016.

[18] Y. Dajsuren, M. Brand, A. Serebrenik und S. Roubtsov, Simulink models are also software: Modularity assessment, 2013.

[19] ECSS-E-ST-70-41C – Telemetry and telecommand packet utilization, 2016. [Online]. Available: https://ecss.nl/standard/ecss-e-st-70-41c-space-engineering-telemetry-and-telecommand-packet-utilization-15-april-2016/.

[20] J. Miller, J. Mukerji, MDA Guide Version, 2003. [Online]. Available: https://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf.

[21] S. R. Chidamber und C. F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Trans. Softw. Eng., Bd. 20, pp. 476-493, #jun# 1994.

[22] C. F. J. Lange und M. R. V. Chaudron, Managing Model Quality in UML-Based Software Development, in Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice, Washington, 2005.

[23] J. Muskens, M. Chaudron und C. Lange, Investigations in applying metrics to multi-view architecture models, in Proceedings. 30th Euromicro Conference, 2004., 2004.

[24] S. Henry und D. Kafura, Software Structure Metrics Based on Information Flow,IEEE Transactions on Software Engineering, pp. 510-518, Sept 1981.

[25] B. Du Bois, C. F. J. Lange, S. Demeyer und M. R. V. Chaudron, A Qualitative Investigation of UML Modeling Conventions, in Models in Software Engineering: Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers, T. Kühne, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 91-100.

[26] S. Efftinge, M. Völter, oAW xText: A framework for textual DSLs, in Version 1.1, September 22, 2006.

[27] M. Perrotin, T. Tsiodras, J. Delange, J. Hugues, TASTE Documentation v1.1, 2012.

[28] V. Rybakov, Linear temporal logic with until and next, logical consecutions, Annals of Pure and Applied Logic, Bd. 155, pp. 32-45, 2008.