Something is Rotten in the State of Documenting Simulink Models

Vera Pantelic, Alexander Schaap, Alan Wassyng, Victor Bandur and Mark Lawford
McMaster Centre for Software Certification, McMaster University,
1280 Main St W, Hamilton, ON L8S 4K1, Canada
{*pantelv, schaapal, wassyng, bandurvp, lawford*}*@mcmaster.ca*

Keywords: Model-Based Development, Simulink, Software Documentation, Embedded Software, Software Requirements Specification, Software Design Description

Abstract: In this paper we draw on our experience in the automotive industry to portray the clear need for proper documentation of Simulink models when they describe the implementations of embedded systems. We effectively discredit the "model is documentation" motto that has been hounding the model-based paradigm of software development. The state of the art of documentation of Simulink designs of embedded systems, both in academia and industrial practice, is reviewed. We posit that lack of proper documentation is costing industry dearly, and propose that a significant change in development culture is needed to properly position documentation within the software development process. Further, we discuss what is required to foster such a culture.

1 Introduction

We have witnessed first-hand, in industry, and especially while working with large automotive original equipment manufacturers (OEMs) and automotive suppliers, the detrimental effect of non-existent or poor software documentation. Other researchers also have traced the poor quality of software engineering practices in industry to the lack of appropriate software documentation (Sousa and Moreira, 1998). Some, most notably David Parnas, have identified improper software documentation as *the* prime cause of industry's erratic record regarding software quality (Parnas, 2011). While the cost of lack of proper documentation is hard to measure, we provide anecdotal evidence that illustrates its effects in development and maintenance of production-scale embedded systems in Simulink/Stateflow. We believe that these effects often translate to large financial losses for companies.

"Models are documentation" has followed model-based design (MBD) since its early phases, primarily promoted by tool vendors (e.g. (Barnard, 2005; Brück et al., 2002)). However, our experience clearly indicates that any developer that has ever been tasked with reverse-engineering a large industrial-scale Simulink model understands all too well that a Simulink model is *not* complete enough documentation. A Simulink model is only one view of the system being implemented – albeit an executable one. However, different stakeholders need different views: the developer of model *A* needs to understand the model's inner-working, while the developer of model *B*, which interfaces with *A*, needs to know only the interface of *A*. It is important to note here that we mean *both the syntax and semantics* of the interface. Simulink provides the syntax of the interface of *A*. *Additional* documentation is necessary to provide the semantics. In addition, the developer of *A* needs to document rationale of design decisions made in the implementation of *A*. The crucial problem is that this *additional* information is rarely documented, even though history teaches us how important it is (Sagoo et al., 2014).

From our experience, many developers appreciate the benefits of good software documentation. Refactoring large industrial-scale Simulink models is practically impossible without good documentation. In fact, developers have asked us on different occasions to help them document Simulink models as – or, rather, because – they were going through a gruelling process of refactoring another engineer's model. Further, the same developers cited "lack of documentation culture" as the main cause for not creating and/or properly maintaining software documentation – production and proper maintenance of documentation is not part of the companies' values and expected behaviour of their developers. Documentation efforts are not appreciated, and thus seldom rewarded, even during (intense) reverse-engineering activities when their benefits become painfully obvious.

This paper surveys the state of the art in the documentation of Simulink implementations of embedded systems, in both academia and industry. In particular, we present our experience distilled from collaborations with multiple automotive OEMs and suppliers to illustrate the industrial need for good documentation of Simulink models of embedded controllers. Also, we address existing work and existing tool support, and discuss their possible extensions that could provide proper support for the documentation process. Therefore, our position is that proper Simulink model documentation would be a cost-effective solution for proper maintenance of production-scale Simulink models. However, this would require a large cultural change in how the industry perceives documentation. This change would require investment in providing technical support for efficient creation and maintenance of documentation, including proper tooling to support these activities.

The rest of this paper is structured as follows: Section 2 describes what documents would be useful companions to Simulink models and why. Section 3 presents some of our relevant experience with industrial Simulink models and explains why these documents are crucially important for large modeling efforts. Section 4 reviews currently available support for documenting Simulink models and Section 5 describes our position on the need for cultural change regarding software documentation. We conclude in Section 6.

2 Documenting Simulink Models: What and Why?

The application layer of embedded automotive software is often designed in Simulink/Stateflow. For large controllers, the design consists of dozens of large Simulink/Stateflow models, where the largest models contain hundreds of thousands of blocks and a few tens of thousands of subsystems. Each Simulink model is typically assigned to one developer. Therefore, a Simulink model corresponds to a software module, in the spirit of the module's definition as a responsibility assignment (Parnas, 1972).

In this context there are many reasons why software documentation is essential, and they all revolve around communication. Every document communicates a different aspect of the software, offering a useful abstraction to its users. These aspects need to be communicated between domain experts, software developers, managers/supervisors, reviewers, testers, maintainers, certification authorities, and users. Documentation is critical not only for communication in the forward development process but also during maintenance. In particular, *traceability* is now recognized to be essential in achieving and maintaining safety, security and dependability (Mader et al., 2013). Traceability includes bi-directional links between system development artifacts, as well as between entities in development and verification, and between system development documents and assurance cases. Without adequate traceability, change impact analysis becomes fragile, and incremental development a nightmare. Thus, our documentation must include the capability to record links between all these entities – far beyond what is achievable in Simulink alone. In regulated industries, documentation is needed for compliance with relevant standards. In particular, in the automotive industry proper software documentation is required by ISO 26262 (ISO, 2011). However, the standard imposes almost no requirements on the content and format of the documents. This is not surprising – software standards typically fail to do so.

Just as in traditional software development, when developing software in Simulink/Stateflow, two documents are crucial (Bialy et al., 2017; Schaap et al., 2018). The first is the model's *Software Requirements Specification (SRS)*, specifying the model's black-box behaviour. The document acts as a contract between the model's developer and the developers of other models that interact with it. It is a document that dictates what the model should implement. Also, the SRS is a reference document for verification activities, where it can be used as a testing oracle. Most importantly, the SRS describes *what* the software must do at a higher abstraction level than would ever be possible when describing exactly *how* the software will achieve the required behaviour – as is the case in Simulink. As further motivation for SRSs in the automotive domain, ISO 26262 strongly recommends requirements-based testing for electronic-software components, no matter what *Automotive Safety Integrity Level (ASIL)* they are assigned.

The second document is the *Software Design Description*, describing a model's internal design that satisfies its requirements. It captures design decisions and anticipated changes, and is written and used by the model's developers in both development and maintenance. The document structure should reflect the model's subsystem[1] hierarchy, documenting the interface and internal design for each important subsystem (Schaap et al., 2018). As described earlier in Section 1, Simulink provides a syntactic definition of interfaces, but it is important that we also provide a

[1]Simulink uses a notion of subsystem to organize large models hierarchically.

semantic definition of the interfaces. The semantic definition has to be added to Simulink, either locally within the model, or in external documentation. The semantic definition often needs to be supported by rationale, trace-links to requirements, etc. This kind of information is extremely difficult or even impossible to reconstruct years after the initial development of the model. This document is therefore crucially important, as the maintainers are typically not the original developers. Regarding the syntactic definition of the interface of a Simulink subsystem, although contained within the subsystem, this definition does not currently have an explicit representation within Simulink. This is due to the fact that, besides the explicit data flow from and into a subsystem represented using input and output ports, Simulink subsystems can communicate via *implicit* data flow mechanisms – data stores (analogous to variables in traditional programming languages) and Goto/Froms. A proposal has been made on how to explicitly document the syntactic definition of a model's complete interface (Bender et al., 2015).

3 Current Practice is Not Good Enough

In general, software documentation is often outdated, non-existent, poorly written and generally untrustworthy (Parnas, 2011; Lethbridge et al., 2003). The same is true specifically of documentation in MBD with Simulink (Rau, 2002; Ackermann et al., 2010; Bialy et al., 2017; Schaap et al., 2018). Our experience in the automotive industry corroborates this perception. There is a fundamental difference between software engineering and traditional engineering disciplines in that, in software development, there is no immediate *material* cost associated with producing *some* prototype of the final product. In traditional engineering, there is a tangible cost above time and effort in building a prototype. Thus, even before a prototype is attempted, initial analyses and preliminary requirements are documented. Since we want to reap as many benefits as possible from the cost of the prototype(s), lessons learned in terms of rationale, design choices etc., are recorded for later use during the construction of the prototype. However, with software there is the perception that it costs nothing to quickly and sloppily "hammer out" code to see if we can make it work as intended. In software, this initial artifact is the first nail in the coffin of proper documentation. To make matters worse, industrial culture regards this initial quick and dirty prototype as initiative on the developers' part and as a meaningful first step toward a solution. Under this type of work culture, effective documentation is promptly and effectively short-cut, with all subsequent effort invested in evolving this initial artifact.

3.1 Stories

To illustrate the impact of lack of (proper) documentation of Simulink/Stateflow models in industry, we summarize some of our experience gained through interactions with automotive industrial partners. The following stories highlight the large development and maintenance costs attributable to a lack of appropriate documentation.

Anecdote 1 A module of an automotive controller implemented a complicated optimization algorithm in Simulink. During the maintenance phase of the software development life cycle, the module was identified as an efficiency bottleneck, and a domain expert was tasked with refactoring it. However, the model itself was sparsely documented, and additionally, used another algorithm that itself was sparsely documented in the literature. Furthermore, it was not clear why some corner cases were implemented in a particular way: the rationale was lost as the original developers of the model were no longer available. Another complication was that the rationale for the design decisions for using certain Simulink blocks over others was also lost. In the reverse-engineering process, the domain expert had to repeatedly consult an engineer who was the only one in the company with a deep understanding of the model. This is a quite common scenario that we experienced on many occasions: the maintenance efforts typically involve consulting different experts in an attempt to understand the model, resulting in a large waste of valuable resources. In this particular case, the domain expert had spent several months reverse-engineering the module and ended up consulting us for help in documenting the algorithm of interest. To the best of our knowledge, the performance bottleneck was never addressed.

Anecdote 2 A collaboration with an industrial partner included development of the plan for the migration of the existing software to AUTOSAR, the open automotive software architecture (AUTOSAR, 2018). In particular, the network interfaces, including CAN (Controller Area Network), SPI (Serial Peripheral Interface) and LIN (Local Interconnect Network), as well as diagnostics, were non-compliant with AUTOSAR, thus requiring an overhaul of the existing solution, including a new, AUTOSAR-compliant mechanism for communication at the application layer, where Simulink models were used. However, the existing communication/diagnostics mechanisms/blocks at the Simulink level were poorly documented. The lack of proper documentation represented a notable hurdle in the development of the migration plan that instigated numerous interactions with both engineers at the application level (Simulink) and the base software level.

Both stories above clearly depict the costs associated with inadequate documentation. During the software maintenance phase, trying to understand under-documented models leads to waste of valuable resources. Reverse-engineering is notoriously difficult, even for domain experts: a great amount of time and effort in addition to domain-specific knowledge is required to reverse-engineer models. Further, reverse-engineering is not always necessarily completely successful: even if the *how* (which is doable with a lot of effort) and the *what* (which is incredibly difficult to differentiate from *how*) are discoverable, the *why* is likely not. Additionally, some non-functional requirements as well as timing requirements, may not be recovered through reverse-engineering.

3.2 The Model is *Not* the Documentation

Both anecdotes above effectively refute the "model is the documentation" myth. As the stories illustrate, and as we have seen on many other occasions, documentation of the design of a model that captures design rationale is a crucial artifact that, ideally, contains the information that makes reverse-engineering efforts largely unnecessary, or at least significantly easier and more effective. Further, documenting design during the design process enhances the problem-solving process by enabling designers to better understand the problem and have a firmer grasp on different possible solutions.

The "code is the documentation" paradigm that existed (sometimes still exists) in traditional software engineering, is analogous to the MBD catchphrase that the "model is the documentation". Years ago, Parnas (Parnas, 2011) discredited it by arguing that different views, with different abstractions from detailed code, are needed. The same holds for models.

4 Current Support for Simulink Documentation

4.1 Related Work

Academia has shown little interest in documenting Simulink models. There exist several causes for this situation. First, the research lacks exciting theoretical underpinning. Second, industry has not shown much interest. Given the pressures of releasing often and cutting down on development costs, industry often lets documentation efforts lapse.

Rau notes some of the common problems with creating and maintaining software documentation for Simulink models (Rau, 2002), namely, that it is often ambiguous, inconsistent, non-existent, incomplete, and tedious to create. This is attributed to documentation being a separate artifact from the implementation in addition to being created at a different time, it being generally informal and unstructured, with redundancy between both documents themselves as well as between documentation and model. Rau then sets principles for effective production and use of documentation in development with Simulink. First, Rau advocates for an integration between a model and its documentation. MathWorks later provided support for this principle by introducing a special kind of Simulink block: DocBlocks allow one to embed descriptive text in a Simulink model. Further, Requirements Management Interface (RMI) allows linking to external requirements documents to create traceability from the model to the requirements, as Rau advocates. In his own demonstration, Rau uses custom blocks to document signals in order to produce the model/subsystem interface. MathWorks has added fields for e.g. specifying units for subsystem ports, but otherwise the generic Description block property can be used to document signals textually.

The next important principle emphasized by Rau is structuring and formalizing documentation, while offering developers placeholders to fill in (Rau, 2002). While Rau does not elaborate this principle, the contents for both SRS and SDD for Simulink models are discussed in (Bialy et al., 2017). Further, Schaap et al. define the (minimal) contents of the Software Design Description (SDD) for Simulink models and propose tool support to semi-automate the generation of SDDs (Schaap et al., 2018).

4.2 Tool Support

Along with the possibility of embedding documentation within a model, the key is the ability to generate external documents. MathWorks' *Simulink Report Generator* generates reports in various formats from Simulink models at the press of a button. However, the default templates offered by the tool only extract information already present in the executable part of the model – resulting documents consist primarily of block parameter tables and are, consequently, of limited use. Additionally, formatting a resulting report

(MS Word document) can be quite difficult. For example, formatting tables, heading font sizes and numbering, or eliminating unnecessary space between elements such as headers and contents, can be notoriously hard. Furthermore, disabling components (deleting sections) of the template sometimes causes unexpected behaviour. Lastly, there is no search functionality built into the tool's graphical user interface (GUI), which makes developing a larger template tedious. To create traceability to requirements, there is the RMI, but documents must be in specific locations relative to the model. Traceability from external requirements to portions of a model is possible as well, but is even more fragile. While these are the building blocks for the approach advocated by Rau (Rau, 2002), they do not constitute a practical approach on their own since the usability deficiencies will deter most developers.

The approach of (Schaap et al., 2018) prescribes a (customizable) SDD template and provides guidance on its contents. Developers add documentation via customized template-prescribed DocBlocks in either text or RTF format. The reports are then generated via Simulink Report Generator. The benefit of defining the structure is consistency across the organization and less developer effort to produce documentation. Once the DocBlocks have been filled in by the developer, document generation and formatting are fully automated. Also, documentation of syntactic subsystem interfaces is fully automated using the Signature Tool (Bender et al., 2015). This automation allows developers to focus on writing useful contents such as rationale for design decisions. Since the work of Schaap et al. proposes minimal content for an SDD, an organization is likely to want to customize the SDD template. Customization, however, would present a tedious task given the formatting issues with the underlying Simulink Report Generator as the report generation engine.

The tool support for authoring, analysis, and management of requirements is much better than the presented tool support for documenting a model's internal design. Since Simulink R2017b+, MathWorks offers the *Simulink Requirements* toolbox. By default, Simulink Requirements provides a developer with ID, Summary, Description and Rationale columns for requirements; however, custom attributes can be added for sets of requirements. In this way, for example, anticipated changes can be expressed. The requirements can then be linked to tests, models and code. Simulink Requirements allows for requirements to be expressed using natural language only.

4.3 Templates

Like Rau (Rau, 2002), Bialy et al. (Bialy et al., 2017) and Schaap et al. (Schaap et al., 2018), we deem precisely defined content of the SDD as critical to the successful utilization of documentation in software development and maintenance processes. Our experience with industry partners clearly confirms this. In our collaboration with an automotive OEM, we have witnessed a case where, while the basic structure (table of contents) of the SDD was defined, the required content was not precisely specified. For example, the template prescribed that the *architectural view* of a Simulink model be included in the model's SDD. However, the term was not defined anywhere and no requirements were imposed on the section's content, leading to inconsistent SDD documents throughout Simulink models of the same controller. Numerous similar issues lead to an ineffective use of documentation and, ultimately, its abandonment. The inconsistency problem caused by the lack of common understanding of the term *architectural view* was not surprising: after all, there exists no single agreed-upon definition for software architecture (Garlan and Perry, 1995). However, we found out that even some terms that are well rooted and used within the software community, were largely misunderstood. For example, the SDD required rationale for design decisions to be documented. The concept of rationale, however, was poorly understood among the developers: many of them mistook it for the purpose of the subsystem. This is in part due to many of the developers in the embedded software industry lacking software engineering background (Bialy et al., 2017). Organizations ought to document their processes and describe resulting artifacts within internal documents (as standards, guidelines, and/or procedures), including precise definition of all relevant terms. In addition, for example, in the approach of Schaap et al., described earlier, the terms and concepts as defined in a document can be leveraged from within documentation templates and included in the generated documentation, where appropriate.

5 Documentation Culture

Recognizing that documentation in Simulink model development is necessary is only part of the solution to the problem of unmaintainable Simulink models. Developing a sustained standard of documentation that includes conventions and notations that prove useful in all aspects of the development and maintenance of these models is also necessary.

We propose a culture of documentation as an effective way of maintaining such a standard.

We intend “culture” here to take on its usual meaning, as a set of shared beliefs, attitudes, values, etc. within an organization. Because culture is cohesive, a culture of documentation would allow a belief and understanding that documentation is just as important as the artifact it documents to span the organizational hierarchy. For such a culture to develop, support is required on several fronts. Change management is notoriously challenging, and there are some proven approaches to it (Anderson and Anderson, 2010). In support of these well documented approaches, we discuss some aspects specific to our suggestion.

5.1 Coming from the Top

It is vital that managers emphasize long-term company concerns with respect to the development of software intensive products. Consequences of company software development culture are not as intuitive as they may be in other engineering disciplines. For example:

- Managers must foster the idea that quality must be “designed in” from the start. Too often, managers promote too early development of design and even code simply because it is so easy to create software. Creating software in an ad hoc manner is not the same as creating, documenting and using a methodical approach to plan and design high quality software that provides a safe and dependable product. Early “solutions” often turn out to be mirages – the closer we get to them the more we realize that they do not deliver what they promised, and when they do successfully deliver on the required functionality, they are likely to degrade during maintenance simply because they were not designed with adequate foundations to start with. A civil engineer, for example, will not build a structure without first performing a slope stability analysis. Ignoring necessary analyses is devastating in any engineering discipline, but with software the devastating effects are often not realized until too late, and then amortized over the lifetime of the software.
- In the vast majority of industrial cases, software is not maintained by those who created it. An awareness must be developed that newcomers will likely handle the software during its maintenance phase – and the original developers may not be available for discussion! In addition, the implications of software developers having to reverse engineer these artifacts sometime in the future were documented in an earlier section.

On the technical side, the change in culture requires templates, standards, procedures and guidelines to support effective creation, use and maintenance of documentation. We discuss this next.

5.2 Supporting standards, guidelines, templates

Successfully fostering a culture of documentation requires a policy that enforces the production and maintenance of documentation during the software lifecycle. This policy should impose the existence of documentation templates, but as we have seen, templates alone are not enough. While templates prescribe the format and content of the documentation, they are not nearly as useful if, for example, developers are unaware of the terminology used in the documents; the notations to be used; or the process of documentation maintenance as part of the change management process. Therefore, internal documents (be they guidelines, standards, procedures, or a combination thereof) are needed to precisely define the development processes including the software documentation activities and the artifacts to be generated by their application. Process documents rarely, if ever, provide enough detail in the process steps to ensure that the results of applying the process consistently achieve the quality attributes desired. There are many reasons for this. Two of the most important are: i) it is difficult to document process steps to that level of detail; and ii) most companies like to leave things less well-defined so that developers have more freedom to do what they think is right in a given situation. One way of combating this that seems acceptable to most developers, is to be more prescriptive in the content and format of the generated documentation.

5.3 Rigour

In other engineering disciplines, documentation typically uses mathematics to specify and describe systems precisely; unfortunately, this is not the case with software documentation where natural language is prevalent. However, natural language suffers from ambiguity. Many methods for formal specification of requirements exist, for instance tabular expressions (Wassyng and Janicki, 2003). We advocate the use of mathematics to make documents more precise and leverage some advanced verification tools. Formal requirements can be automatically checked for consistency and completeness. Crucially, they can be used as a basis for automatic test generation and for formal verification of Simulink designs against requirements. In fact, advanced commercial tools exist for automatic test generation and formal verification of Simulink models (e.g., MathWorks' *Simulink Design Verifier (SDV)* and *Reactis* by Reactive Systems).

5.4 Automate, automate, automate

Automation of routine tasks is essential to the development and maintenance of software (Hunt and Thomas, 2002). Proper tools are essential in supporting documentation efforts. Existing tool support for documentation of Simulink models is not yet adequate. For example, Simulink Report Generator needs to undergo major improvements to be used as a proper engine for automatic generation of reports from defined templates. The tool described by Schaap et al. (Schaap et al., 2018) is an academic tool that leverages Simulink Report Generator and needs to be improved to provide easier template customization as well as a GUI for, at least, configuration.

MathWorks' Simulink Requirements toolbox can also be enhanced. As mentioned before, adding fields for anticipated changes would improve the quality and usefulness of the requirements in preparing a design that may be more robust with respect to future changes. Simulink Requirements can store requirements only in ASCII text or MS Word. However, if requirements are formalized using Simulink/Stateflow, as Simulink/Stateflow models outside of Simulink Requirements, they can be linked to their textual representation in Simulink Requirements using traceability links provided by the combination of the Simulink Requirements and Simulink Test toolboxes. Tests could be derived using Simulink Test or Simulink Design Verifier (SDV). SDV tests can be imported into Simulink Test, and could then be linked to requirements and designs.

6 CONCLUSIONS

We have presented empirical evidence that industrial practice demonstrates a need to properly document Simulink models. The anecdotes we presented may be few, but they are drawn from more than five years of experience working closely with automotive industrial partners on their Simulink models. They are representative of what we observed in general. We also presented the case for a cultural change that we believe is required to improve both the *quality* and *status* of documentation in the MBD process in industry. These changes require substantial resources, as well as direction and commitment from management. We believe that the required resources and dedication will pay off in the long term. As immediate future work, a systematic approach would examine the Simulink models documentation practices in the automotive industry and help define more precise documentation requirements. This work would pave the way to creating effective documentation standards, defining more concrete actions in implementing the documentation culture, and developing better tool support.

REFERENCES

Ackermann, C., Cleaveland, R., Huang, S., Ray, A., Shelton, C., and Latronico, E. (2010). Automatic requirement extraction from test cases. In Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Rosu, G., Sokolsky, O., and Tillmann, N., editors, *Runtime Verification*, volume 6418 of *LNCS*, pages 1–15. Springer Berlin Heidelberg.

Anderson, D. and Anderson, L. A. (2010). *Beyond Change Management*. Pfeiffer, San Francisco, California.

AUTOSAR (2018). AUTOSAR – enabling innovation. `https://www.autosar.org`. [Online; accessed November, 2018].

Barnard, P. A. (2005). Software development principles applied to graphical model development. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, San Francisco, CA, USA. American Institute of Aeronautics and Astronautics.

Bender, M., Laurin, K., Lawford, M., Pantelic, V., Korobkine, A., Ong, J., Mackenzie, B., Bialy, M., and Postma, S. (2015). Signature required: Making Simulink data flow and interfaces explicit. *Science of Computer Programming*, 113, Part 1:29–50.

Bialy, M., Pantelic, V., Jaskolka, J., Schaap, A., Patcas, L., Lawford, M., and Wassyng, A. (2017). Software Engineering for Model-Based Development by Domain Experts. In Griffor, E., editor, *Handbook of System Safety and Security*, pages 39–64. Syngress, Boston, 1st edition.

Brück, D., Elmqvist, H., Mattsson, S. E., and Olsson, H. (2002). Dymola for multi-engineering modeling and simulation. In *Proceedings of Modelica*, volume 2002. Citeseer. Online at `https://www.modelica.org/events/Conference2002/index_html`.

Garlan, D. and Perry, D. (1995). Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274.

Hunt, A. and Thomas, D. (2002). Ubiquitous automation. *IEEE Software*, 19(1):11.

ISO (2011). ISO 26262: Road vehicles – functional safety, International Organization for Standardization (ISO).

Lethbridge, T. C., Singer, J., and Forward, A. (2003). How software engineers use documentation: the state of the practice. *IEEE Software*, 20(6):35–39.

Mader, P., Jones, P. L., Zhang, Y., and Cleland-Huang, J. (2013). Strategic traceability for safety-critical projects. *IEEE Software*, 30(3):58–66.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Parnas, D. L. (2011). *Precise Documentation: The Key to Better Software*, pages 125–148. Springer Berlin Heidelberg, Berlin, Heidelberg.

Rau, A. (2002). Integrated specification and documentation of Simulink models. International Automotive Conference.

Sagoo, J., Tiwari, A., and Alcock, J. (2014). Reviewing the state-of-the-art design rationale definitions, representations and capabilities. *International Journal of Design Engineering*, 5(3):211–231.

Schaap, A., Marks, G., Pantelic, V., Lawford, M., Selim, G., Wassyng, A., and Patcas, L. (2018). Documenting Simulink designs of embedded systems. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '18, pages 47–51, New York, NY, USA. ACM.

Sousa, M. J. C. and Moreira, H. M. (1998). A survey on the software maintenance process. In *Proceedings of 1998. International Conference on Software Maintenance*, pages 265–274. IEEE.

Wassyng, A. and Janicki, R. (2003). Tabular expressions in software engineering. *Proceedings of 2003. International Conference on Software and System Engineering (ICSSEA'03)*, pages 1–46.