A Survey about Modeling Guidelines for Simulink and Stateflow

Gerald Stieglbauer

31st July 2006

University of Salzburg, Department of Computer Sciences,
Jakob-Haringer Str. 2, 5020 Salzburg, Austria
gerald.stieglbauer@cs.uni-salzburg.at

Abstract

The importance of model-based development of software for embedded systems has significantly increased in the last couple of years. At the outset of this trend, various modeling tools based on different model abstractions were available on the market. Originally, most of these tools had not been specifically designed for embedded software development. Simulink and Stateflow[1], which have been primarily designed for simulation, are maybe the most popular examples. Because of the promising philosophy of model-based software development, many modeling tools including Simulink and Stateflow have been adopted for embedded software development as well. Due to their popularity and availability, Simulink and Stateflow have established themselves as de-facto standards in this respect. However, because of their original background, the extensive modeling features of Simulink and Stateflow are not always well-suited for embedded software. Consequently, it is often hard to review corresponding models against quality requirements of embedded systems. In order to overcome this lack, several modeling guidelines for Simulink and Stateflow have been developed over the last years. Unfortunately, most of these guidelines have been developed for in-house use and are not publicly available. This article collects the rare exceptions of that trend by presenting a survey about publicly available modeling guidelines for Simulink and Stateflow. Currently, as a result of the increasing market pressure to develop high secure and reliable software for embedded systems, several working groups aim at the development of a universal modeling guideline document. These efforts are sketched by this report as well.

[1]Simulink and Stateflow are registered trademarks of The MathWorks, Inc.

1 Introduction

Embedded systems are going to change our daily life continuously. Nearly every day, new innovative products are presented to the market. Promising advancements in the hardware domain related to keywords like system-on-a-chip, VLSI design, hardware/software co-design, etc. are a contributing factor to those products. Nevertheless, in various cases, the innovative character of embedded applications is not entirely caused by the hardware itself. Especially in domains that have to deal with long time to market periods (like the automotive industry), software is the more flexible and mass production time/costs invariant part. Although the hardware and its continuous advancement form up an important foundation for the improvements in these application domains as well, most innovations are still software-based.

Embedded systems are found in cellular phones, other mobile devices like MP3 players and handhelds, washing machines, up to cars, airplanes and medical systems. As a result of the huge number of different applications, there is a large array of significantly different requirements concerning the development process of embedded systems.

At the time embedded systems were introduced, it was quite usual to program them by low-level assembler code. As the applied micro-controllers became increasingly powerful and the costs for memory and storage fell significantly, more complex applications became tangible from a technical point of view. In parallel, the dropping hardware costs made embedded systems more attractive for mass production since the efforts for software reproduction were negligible low.

Soon, a shift from assembler code to higher level programming languages became necessary. Due its high availability (user acceptance and experiences, compilers, etc.) and its low level features (like pointer arithmetic), the general purpose programming language C had great success in this respect. Another example is the language Ada, which was rather seldom applied in cases where safety issues were more relevant. Consequently, many embedded systems related issues had influenced the language design of Ada.

As indicated above, different domains of applications adhere to different levels of quality and reliability requirements as well as safety requirements. Additionally, the functional capability of embedded systems has exploded in the last couple of years. Today, embedded systems are composed of millions of lines of code, which is distributed over dozens of electronic control units (ECUs). To master the development of such applications, new software engineering paradigms have to be introduced. As only one example, the MISRA consortium delivered important standards and guidelines for the development process within the automotive industry. Since the programming language C has not been specifically designed for embedded systems, the guideline set of the MISRA-C standard [29] restricts the usage of C in order to adapt the language for embedded systems related issues and to avoid many sources of design weaknesses and software failures in advance.

Now, MISRA-C is a widely accepted standard even for code generation. A similar standard applicable to code generation in the safety-critical domain is the IEC 61508 [20] standard.

As an alternative to the MISRA standard or similar approaches, which are applied on existing technologies, new abstraction models were developed in order to simplify the software development process. This shift is comparable to the shift from assembler code to higher level programming languages designed for embedded systems (e.g. Ada). In the area of time critical software systems, for instance, the languages Esterel [5], Lustre [18] and Signal [4] have quite success. Additionally, the Timing Definition Languages (TDL) [36] currently tries to establish itself in this field as well by introducing a new abstraction model [24].

These new software development technologies are often subsumed by the term *model-based software development*. Although the already mentioned model-based languages have a textual representation of their abstraction model, this has not to be the case in general. Quite the contrary, many development tools were developed, which offer a graphical representation of their programming models (TDL [35] and the bundle SCADE/Lustre offers even a textual and a graphical representation). Examples of these tools are UML tools (like Rhapsody [17] or ARTiSAN [37]), LabView [22], Ptolemy[9], Simulink [28] and Stateflow [27]. Simulink and Stateflow, which have been originally developed for simulation, have established themselves as de-facto industry standards in many application domains. Similar to C, popularity and availability have been the key issues for their success in embedded system development.

Nevertheless, it seems that history is repeating at this point: Many of the mentioned tools have not been designed exclusively for the embedded domain, beeing equipped with huge amounts of modeling possibilities leading to models that differs in efficiency and other quality aspects. Consequently, the shift from assembler to C is directly comparable to the shift from C to Simulink/Stateflow: Similar to the MISRA-C standard, modeling guidelines have been developed or are currently under development in order to adapt the modeling tools for embedded software development and increase various quality attributes of such models.

This report presents a survey about modeling guidelines for Simulink and Stateflow. In the next chapter, several publicly available guidelines are presented by explaining their historical background, application domain and tool support. Additionally, future trends in the development of such guidelines are sketched.

2 Overview of Available Simulink/Stateflow Guidelines

In the last decades, several software engineering terms have been introduced with the intention to support software engineers in designing, structuring and

implementing of their applications. So-called *software patterns* and *framework cookbooks* are two common examples of such terms. Originally, the term *pattern* has been introduced by Christoph Alexander [1] in order to describe patterns in buildings and towns. Nevertheless, the term was adopted by software engineers, especially in the field of object-oriented programming. Over the years different kinds of software patterns have evolved. A reasonable classification by three major pattern types might be the following: *Architectural patterns* [34] give hints about global structures of an application but do not define details for implementation. The most famous usage of the term pattern in the software domain is introduced by Gamma et al. with the book *Design Patterns* [16]. The patterns presented within this book are not as global regarding the structure of a certain application than the architectural patterns, but provide reasonable solutions for common software design issues. Design patterns are usually accomplished with many implementation details like class diagrams and coding examples. Finally, *coding patterns* [25] are usually not related to a specific application and its inner structure. Instead, they focus on the basic use of language constructs in order to avoid the adoption of language constructs in an unintended way or to restrict the specific language to a subset of language constructs. The MISRA-C guidelines [29] are one example of the latter case. For instance the usage of the C's pointer construct is restricted so that a wrong or careless use[2] of pointers is prevented (e.g. by proscription of pointer arithmetic or using more than two levels of pointer indirection). *Cookbooks* often focus on the efficient and correct use of frameworks. The Java tutorial [8] is a classical example of such a cookbook.

In this document, we use the term guideline as an umbrella term for patterns, cookbooks and other related terms. This is motivated by the content of the presented guidelines, which only roughly distinguish between the classical terms mentioned above. In the ensuing presentation, the association of a certain rule subset to one of the above mentioned terms is outlined whenever possible.

2.1 Motivation of Modeling Guidelines

In this section, we want to clarify the main intentions of the use of guidelines for model-based development, especially in the context of Simulink and Stateflow. An incomplete list of such intentions is the following:

- Evolving a *common language* for modeling constructs and patterns to simplify explanations about a certain model
- Developing a *common modeling technique* to share models between users
- Giving hints, which *modeling alternative* is the best suited for a specific application

[2]Although a crystal clear definition found in [23], the C pointer construct is susceptible to programming errors, unfortunately.

- Defining a *subset of modeling elements* that have been proved to be safe
- Defining a set of *instructions for model elements* to aim some specific code generation requirements
- Defining a *library of modeling elements* that should be used in a certain context
- *Sharing experiences* gained by senior users and former projects
- *Increasing the quality standards* of models
- Increasing *readability, structuring* and *enhanceability* of models
- *Simplifying model debugging*
- etc.

A *common language* for patterns is useful when the software designer and the programmer try to discuss the architecture and elements of a certain application. Of course, a common modeling language focuses more on architectural and design patterns than on coding patterns and cookbooks: Coding patterns often present a special implementation detail, whose name is rather generic depending on the current application. Cookbooks are more a reference book, which tell you how to implement a special part of an application. Architectural and design patterns, however, can appear at several places within an application and - in case of model-based development - usually group a set of modeling elements for a certain purpose. By giving this set a certain name, more structured discussions about the model become possible. Some modeling guidelines have separate sections that focus on presenting reference implementations of certain patterns. However, this is not the general case and therefore reference implementations are often intertwined with guidelines that tend to be more an equivalent to coding patterns.

A *common modeling technique*, which, of course, can be derived from a *common language* for modeling constructs and patterns, does not only support discussions between experts but also the exchange and reuse of models and parts of them, respectively. However, model sharing at that level is not restricted to a specific modeling tool alone. If two tools are based on a similar abstraction model, it is even possible to share models with a minimum of adaptations. Modeling guidelines that aim at that goal usually come with an (abstract) definition of a model library, whose elements implement concrete patterns or restrict model elements in a proper way.

Usually, modeling tools offer several possibilities of solutions for a certain problem. However, the efficiency, side-effects, and other quality attributes of these solutions differ significantly. In the most extreme case, the user is able to apply modeling elements to a certain problem even if their semantic was originally

not intended to solve that problem. In many cases it is hard to find out which of the *modeling alternatives* offers the best suited solution. For a modeling novice, the try and error method is often the only way to find out which solution is better than another. However, this process is very time consuming. Additionally, even if the user has come to a decision, there is still no guarantee that the chosen solution is really the best that is available. Guidelines, which document a good solution for a given problem, are able to reduce these efforts significantly. Such guidelines usually present a modeling pattern, which is based on long time experiences of former projects.

As mentioned before, the MISRA-C standard provides guidelines on how to use a subset of the programming language C in order to avoid common programming mistakes. An alternative way is to design a new programming language, which is more restrictive but offers enough flexibility concerning the given requirements. The language Ada is one such example, but there are some C dialects available as well, which are based on reduced sets of language constructs. When modeling tools were developed, pioneer work was done coming with many advantages and disadvantages. Of course, disadvantages were caused as well by the lack of experiences with the new possibilities of the tools regarding concrete solutions. Therefore, modeling constructs have been introduced, which turned out to be rather counterproductive than helpful. For instance, in Simulink it is possible to define a block priority in order to influence the block update order. This can lead to models, whose behavior is hard to understand. Furthermore, the addition of some function blocks can influence the behavior of others, which is not obvious indeed. The reason for that are design lacks during the development of the tools but also the continuous enhancements of many modeling tools (which is often caused by the efforts of fulfilling certain customer requirements), which has unforeseen side-effects. However, to ensure backward compatibility, these enhancements cannot be revoked or redesigned easily. Therefore, modeling guidelines were introduced, which prescribe a restricted use of modeling constructs (i.e. a *subset of modeling elements*) and thus avoid the creation of ambiguous models.

A related issue is a certain use of code generators, which are available for many modeling tools, in order to meet some quality requirements of the generated code. For instance, it is important for applications in the automotive domain that the generated code adheres to the MISRA-C standard. Modeling guidelines, consisting of *instructions for modeling elements*, can help to create models whose transformations into real code consider such requirements [3].

Many modeling tools offer the possibility to create *libraries*, which contain *modeling elements* (e.g. function blocks) with a pre-defined functionality. These libraries are usually employed to organize sets of often used modeling elements. Additionally, they can be applied as well to enhance or restrict the basic modeling elements. For instance, code generators may need additional information for code generation or restrictions may be needed for exchanging models between different

modeling tools. Consequently, libraries could be seen as modeling guidelines in a further sense. Frequently, libraries are delivered with additional guidelines on their efficient use.

Concluding, it is to say that modeling guidelines support the design of efficient, well-structured and unambiguous models. Additionally, they significantly reduce development efforts by preventing the user from a recurrent invention of the wheel and by reusing experiences of former projects. It is important that the user does not see these guidelines as an extra burden that limits its creativity but as a helping hand in taking advantage of the model-based development. Currently, many companies possess and develop such guidelines mainly for in-house use [12]. Consequently, many guidelines are unfortunately not publicly available because of judicial and competition issues. Rethinking this issue and a company-wide cooperation with the aim of making reference guidelines publicly available have a great potential for all participants. Of course, it is important that such guidelines do not overspecify everything and allow project specific enhancements or modifications [10]. Guidelines should remain as flexible documents: If modeling tools are updated by new features or new requirements are established in a certain domain, the guidelines should be frequently revised in parallel. The next section presents a list of (publicly) available guidelines for Simulink and Stateflow.

2.2 Available Guidelines for Simulink and Stateflow

This section presents a short list of guidelines for Simulink and Stateflow that are either publicly available or are accessible by joining corresponding communities. It was difficult to decide how big the set of rules and patterns of a certain guideline document had to be in order to be relevant for inclusion in this list. This report focuses on guideline documents that either consist of dozens of rules and patterns or have a widely accepted contribution in certain domains. As an example of document not included here, [6] claims to provide some form of a guideline to port some benefits from the object oriented software development process to certain Simulink models by providing a straightforward guideline about implementing an object in Simulink.[3]

Ford powertrain guidelines (FPG, published in 1999) One of the first guideline documents for Simulink was published by Ford [14][4]. The guidelines emerged from prior studies done by Ford during powertrain development for Ford's Puma Diesel variant and other related projects. Generally, the document

[3] Although the document presents how to implement methods and initialization in Simulink, it does not explain how the most important object-oriented principle could be realized, namely class inheritance.

[4] `http://vehicle.berkeley.edu/mobies/papers/stylev242.pdf`

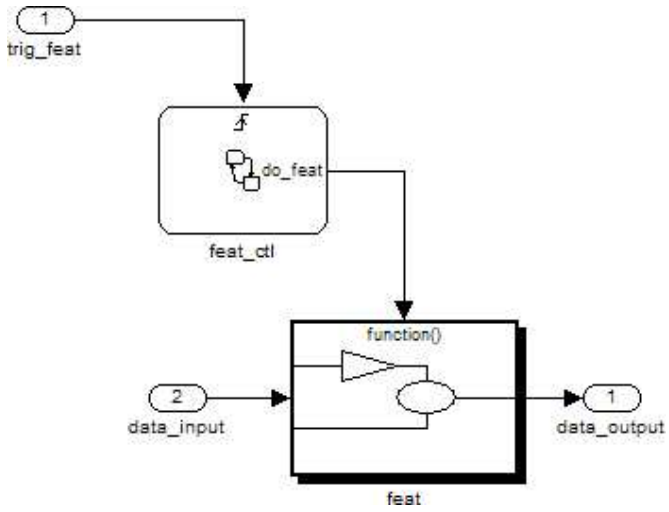

Figure 1: Schematic model of the relationship between a C-spec and a P-spec.

aims to clarify how Simulink and Stateflow have to be used in order to define (powertrain) controller functionality. The FPG is structured in several chapters, each containing different types of guidelines. The first chapter is about designing top level models for a specific (powertrain controller) feature. These top level models are called *context diagrams*. For instance, all data inputs and outputs as well as events of the feature that has to be implemented are defined in this diagram. In addition, the context diagram is then structured in further parts according to a certain scheme (e.g. structuring the feature into parts that are executed at different rates). This scheme organizes the model in a standardized way. Consequently, the guidelines found in this chapter can be easily categorized as an architectural pattern. In addition to the pattern itself, a lot of naming conventions are defined here as well. Since the purpose of these top level models is thought for organization and structuring issues only, no feature implementation is explained at this point.

In FPG, features are referred as so-called *P-specs* (process specifications). In general, a P-spec is implemented either by a Simulink or by a Stateflow diagram. By contrast, the so-called C-specs (control specifications) are defined by a Stateflow diagram only. C-specs are used to control the execution (e.g. a certain execution rate) of a specific P-spec. Figure 1 shows the general form of a C-spec (implemented by a Simulink Chart block) and a P-spec (implemented by a Simulink Triggered Subsystem block). The guideline that adheres to that figure is a classical example of a design pattern.

After explaining the top level of the model structure of a certain feature, the definition of P-specs by Simulink blocks are defined by a further chapter. For instance, it is explained how a delay should be implemented or when and how data

has to be merged by the use of Merge blocks. According to our categorization, this chapter could be placed somewhere between design and coding patterns. Finally, lots of naming conventions are explained here as well.

The next two chapters of the FPG present guidelines applicable to Stateflow diagrams only. The first focuses on guidelines about the definition of flow charts (e.g. how an if-then-else construct or control loops have to be modeled in an unambiguous way by using a top-down layout principle). The second chapter is about rules for Stateflow diagrams that hold additional states, i.e. state chart diagrams. Similar to the first chapter, the model layout plays an important role in order to avoid unreadable and even ambiguous models. The latter is possible, since Stateflow incorporates many graphical issues (including the layout) to conclude to a specific semantic. Each chapter is applicable to the development of C-specs as well as P-specs and both are classified to consist of a set of coding patterns. This characterization is underlined by the fact that the guidelines are formulated more application independent and modeling tool (i.e. Stateflow) centric.

Additionally, the FPG presents a set of standardized library components (block sets) in a separate chapter, which aims to standardize common used functions needed for the development of powertrain and related applications. Finally, chapters about how accessing data dictionaries in Simulink and guidelines for model initialization round out the document.

As mentioned in the previous section, the user acceptance of guidelines plays an important role. In order to ensure the acceptance of a certain set of guidelines, it is essential that the modeling engineer understands the motivation of each guideline. Consequently, each guideline document should explain the benefits and constraints of the various rules. In addition, each guideline should be formulated in a compact and easy-to-understand way. Unfortunately, none of these attributes are accomplished sufficiently well by the FPG. For instance, it is not motivated why the model structure of a specific P-spec is better than alternative approaches. Moreover, even if it seems to be natural to implement C-specs in order to control the rate of execution of a certain P-spec, Simulink provides alternative modeling approaches (e.g. by the use of Unit Delay blocks). However, the document neither explains the benefits of the presented approach nor mentions the existence of modeling alternatives. Also, the FPG consists of lot of text instead of short and compact explanations. Furthermore, the explanation of terms (like C-specs and P-specs) is intertwined with the guideline texts. Consequently, a user has to read certain guidelines (which may be of no interest to the user) in order to understand the terms employed in other guidelines.

To sum up, the Ford powertrain guidelines are important in their pioneering role, being one of the first publicly available guidelines for Simulink and Stateflow. They provide a lot of information and inspiration for designing efficient, unambiguous and well-structured models. However, they are sometimes too in-house

specific and the motivation of some guidelines is not clarified. They have apparently influenced other guidelines like the MAAB and J-MAAB guidelines, which are presented in some of the next paragraphs, since many topics and guidelines are reused there in revised versions.

MSR-MEGMA library (first published in 1999) At first glance, the MSR-MEGMA library has marginally to do with guidelines, since it consists mainly of standardized modeling functions, while concrete modeling rules play an underpart. However, this library is mentioned here because its purpose is closely related to one major intention of modeling guidelines: model exchange. A major goal of the MSR-MEGMA project[5] is to enable model exchange between different modeling tools [38], more exactly between the tools Simulink, ASCET-SD and Systembuild. These tools are based on similar model abstractions but differ in details, which are caused by different domain focus and required flexibility of the tools. However, since many software projects use (or have used) the different tools to implement different requirements[6], the industry was interested in the proposed model exchange in order to make interactions of the various models possible. The MSR-MEGMA project is part of MSR (Manufacturer Supplier Relationship) and is a joint research work involving Bosch, BMW, Daimler-Chrysler, Hella and Siemens as leading participants.

The major output of the project is the MSR-MEGMA library [30], which was first published in 1999 and was continuously refined until 2001. Similar to the library presented by the Ford guidelines, the MSR-MEGMA library defines standard blocks, which are commonly used in the automotive domain and are implementable in the mentioned modeling tools. Models that are mainly based on such blocks should be easily convertible between the different tools without any losses. Modeling guidelines, however, cannot be factored out entirely from using that blocks and have to be applied from two points of view: First, model recognition by the user after a conversion plays an important role. With the user observance of modeling guidelines, whose compliance are recognized by the conversion tool as well, the result of the conversion can be improved significantly. Second, not all basic blocks provided by the modeling tools are reimplemented by the MSR-MEGMA library. For instance, the if-then-else construct should be still modeled by using blocks from the standard modeling tool library. In this case, guidelines help to use these standard blocks in every modeling tool in a standardized way, i.e. by defining a subset of block options, which is available in every modeling tool and is one basic requirement of a successful model transformation.[7]

Although there has been no news from the MSR-MEGMA project[8] since the

[5] `http://www.msr-wg.de/megma/`

[6] At least at the time the MSR-MEGMA library was introduced.

[7] For a list of available tools and library implementations see [30] as well.

[8] However, the MSR community is still active.

announcement of the final library in 2001[9], the project illustrates a successful combination of guidelines and model library with the purpose of software standardization and model exchange.

MAAB guidelines (published in 2001) In 2001 MathWorks presented a document entitled *Controller Style Guidelines for Production Intent Using MATLAB®, Simulink® and Stateflow®* [26][10]. This is better known by the title *MAAB guidelines* or *MAAB style guides*. MAAB is an abbreviation for *Matlab Automotive Advisory Board*. MAAB was founded by Ford, DaimlerChrysler and Toyota in 1998 with the purpose to coordinate the feature requests by key customers of MathWorks. The MAAB community is closely related to the *International Automotive Conference* (IAC) which is held by MathWorks once per year[11]. Now MAAB has dozens of members consisting of various automotive OEMs and suppliers.

The MAAB guidelines are intended to establish modeling rules for the design of controllers in the domain of automotive software development. However, many of the rules are general enough so that the MAAB guidelines (or at least parts of them) are used in other domains (see [13] for instance) and are not limited to the design of controllers only.

The guidelines have evolved by discussions between the MAAB members. Consequently, a lot of in-house experience gained during former projects influenced the selection of the guidelines significantly. The MAAB guidelines are at the moment probably the best source to get a feeling about how the content of the in-house guidelines (which are not publicly available) looks like. Many guidelines already found in the FPG are reused by the MAAB document in a revised form. However, the guidelines build only a foundation for guidelines used in a concrete project and remain a compromise between the MAAB members. In real projects, the generic MAAB guidelines need to be adapted and enhanced by further guidelines and rules. Generally, the document mostly consists of coding patterns. For many guidelines, even this term might be too extensive, since each of them consists of a single proscription of using a certain block in a specific context.

The MAAB community motivates the introduction of the guidelines by several arguments. An incomplete list of these arguments are clean interfaces, uniform appearance of models, code and documentation, reusable models, hassle-free exchange of models, cooperation with subcontractors and so on. Furthermore, the community claims an increase of functionality, quality, safety, reduced time-to-market and cost reduction [26].

[9] In between, Matlab/Simulink is the clear market leader compared with the tools ASCET and MATRIXx/Systembuild.

[10] `http://www.mathworks.com/industries/auto/maab.html`

[11] Traditionally the conference location alternates between USA and Germany.

ID: Title	db_xxxx: Title of the guideline (unique, short)
Priority	One of mandatory / strongly recommended / recommended
Scope	One of MAAB / DC / DC-POWERTRAIN / DC-CHASSIS / FORD / FORD-POWERTRAIN / TOYOTA / GM
Automation	One of check / correction / possible / none
Prerequisites	Links to guidelines, which are prerequisite to this guideline (ID+title)
Description	Description of the guideline (text, images)
Benefit	Benefit of respecting the guideline
Penalty	Penalty of breaking the guideline
Author	Name of the first author or name of the source document
Last Change	Date of last change, Author of last change

Table 1: Scheme of a guideline (adapted from [26]).

As the guidelines were introduced in 2001 with the version number 1.0, frequent updates of the document were planed, as modeling guidelines should not be a non-moving target [10]. However, this updates did not happen. Instead, many existing in-house guidelines have been influenced by the MAAB guidelines. They have also been extended by further general, updated[12], or domain-specific guidelines, which are not project specific only and whose integration into the MAAB guideline document could have prevented the multiple designs of the in-house revisions. Another reason for the development of in-house guidelines is the fact that the MAAB guidelines are generic by nature and are insufficient for a particular tool chain or code generation [12]. Although the guidelines have been advanced behind the scene, the current version number of the MAAB guidelines is still 1.0. The J-MAAB guidelines, which act as an enhancement to the MAAB guidelines, are presented in the next paragraph and are a rare exception from that circumstance. Furthermore, new efforts have recently started in order to publish a revised version of the MAAB guidelines.

Since this revised version will be based on the original MAAB document, an overview about the latter's contents is presented in the following. Compared to the FPG, the MAAB guidelines are more systematically structured: Each guideline is formulated according a defined scheme. The scheme consists of the items shown in Table 1. A detailed description of each entry can be found in [26]. However, a major improvement of this document over the FPG is presenting the motivation of each guideline. This is done by the field *benefit*. Additionally, the field *penalty* explains what happens if the guideline is broken. With this

[12]Since 2001 many new updates of Matlab/Simulink with a lot of modeling enhancements are released.

information, the user can decide if the guideline is worth to be applied to a current project or not. In addition, this decision is assisted by the *priority* field. The *scope* field indicates which of the MAAB members agree to a certain rule. However, the whole guideline document consists of rules that adhere to MAAB (i.e. to all MAAB members at the time the document was created). The possible entries FORD and FORD-POWERTRAIN indicate that the Ford guideline document has influenced the MAAB guideline document. Furthermore, links to prerequisite guidelines improve the quality of the document's structure and the *automation* field is important for tool developers in order to check an existing model against the modeling rules by the potential use of a model checker tool. Finally, the *description* field contains the real guideline. Compared to the Ford guidelines, the description of the guidelines is kept relatively short and concise. In general, the guidelines of the MAAB document are more general and application independent than many of the FPG. The definition of a top-down modeling hierarchy for implementing a specific feature is resigned entirely.

There are many similarities between the overall structure of the MAAB document and that of the FPG. For instance, both have separate chapters for modeling in Simulink and modeling in Stateflow. Concerning Stateflow, an additional distinction between modeling Flowcharts and Statecharts is applied as well. The MAAB document contains a lot of rules regarding an easy-to-read layout applied to Simulink models as well as to Stateflow diagrams. These rules are usually organized by separate subchapters that focus on general modeling issues. For example, a rule about forbidden blocks in controller definitions is shown in Table 2.

By contrast to the FPG, naming conventions are not intertwined with other guideline definitions but are subsumed by extra subchapters as well. A further improvement of the document's structure is the explicit distinction between a simple rule and a pattern, which is missed within the Ford document. Although both tend to adhere to coding patterns, a further, finer grained classification is introduced. The MAAB document presents modeling patterns for Simulink as well as for Stateflow. Similar to some guidelines in the FPG, patterns in the MAAB document focus on conditions, transition actions as well as on if-then-else and loop constructs.

To sum up, the MAAB guideline document contains generic and widely application independent rules for modeling in Simulink and Stateflow. The guidelines are based on a well structured scheme that allows easy navigation through the document. The clear and simple guideline definitions and motivations support the acceptance and comprehensibility for the user and support the development of tools for an automated validation of models that should adhere to the presented guidelines. However, since the modeling tools have evolved and the project experiences have increased, an update of the document becomes necessary. Therefore, several communities are currently working on such an update.

ID: Title	jm_0001: Prohibited Simulink standard blocks inside controllers
Priority	Mandatory
Scope	MAAB
Automation	Possible
Prerequisites	none
Description	Controller models must only be designed from discrete blocks. *Sources are not allowed:* Signal Generator, Step, Ramp, Sine Wave, Repeating Sequence, Discrete Pulse, Generator, Pulse Generator, Chirp Signal, Clock, Digital Clock, From File, From Workspace, Random Number, Uniform Number, Number, Band-Limited White, Noise. *Continuous blocks are not allowed:* Integrator, Derivative, Memory, Transport Delay, Variable Transport, Delay, State-Space, Transfer Fcn, Zero-Pole *Other blocks, which are not allowed:* Slider Gain, Algebraic Contraint, Manual Switch, Complex to, Magnitude-Angle, Magnitute-Angle to, Complex, Complex to Real-Imag, Real-Imag to Complex, Hit Crossing, Polynomial, MATLAB Fcn, Goto Tag Visibility, Probe
Benefit	Respecting the guideline ensures... • A controller that more closely represents the way it will be implemented.
Penalty	Breaking the guideline... • will not be able to use triggered or enabled subsystems • will not be able to convert to fixed-point
Author	Judy May
Last Change	14.02.2001, Hank Donald

Table 2: Rule about forbidden blocks within a system controller (adapted from [26]).

J-MAAB Guidelines (published 2003) The *Japan Matlab Automotive Advisory Board* (J-MAAB)[13] was founded by several Japanese automotive companies (Toyota, Mazda, Nissan and Honda) in order to promote model-based design. It is designed as an open user community and exists independently from the other MAAB communities (although cooperation is propagated). Currently, it consists of two working groups: the *Verification Guideline Working Group* (works out V-model based processes applied to model-based development with Simulink) and the *Style Guideline Working Group*. The major outcome of the Style Guideline Working Group was the *Simulink StyleGuide* document [21], published in 2003. This guideline document can be seen as a supplement [12] to the MAAB guidelines. With respect to its contents, the document covers some missing topics that can be found in the FPG but not in the MAAB guidelines. Compared to the Ford document, the topics are presented in a purified (to be more application independent) but enhanced manner. The gist of the J-MAAB guidelines consists of architectural guidelines and some design patterns. These kinds of guidelines are covered in the FPG by presenting hierarchical structures in order to implement a P-spec. Similar to the MAAB document, the guidelines found in the J-MAAB document are more generic and flexible. However, the J-MAAB document misses the description of the motivations and benefits of the presented patterns. In addition, no penalty information is provided. The architectural guidelines are based on a four-layer architecture (illustrated by Figure 2), which can be applied by two formats A and B. The two formats differ in the use of a certain layer, i.e. the so-called *trigger layer*. Finally, the content of each layer can be classified as a classical design pattern.

- *Top layer*: In this layer, the I/O of a controller is modeled. If format A is used (i.e. a trigger layer is defined), the controller is modeled by a single Simulink Subsystem block connected to input and output ports *Format B* is less restrictive and the controller function can be separated into further parts. The input is not limited to data only but includes trigger signal as well.

- *Trigger layer* (only if *format A* is used): This layer consists of several Triggered Subsystem blocks that are placed within the single Subsystem block of the top layer. For each execution rate a separate block has to be placed. For instance, if some controller parts should be executed with a rate of 4ms and some with 8ms, two Triggered Subsystem blocks are inserted, each connected to the corresponding trigger signal.

- *Structured layer*: In this layer, the functional parts of the controller are defined by several Subsystem blocks (i.e. by putting the block into the Triggered Subsystem of the trigger layer if format A is used or within the

[13] `http://j-maab.cybernet.jp`

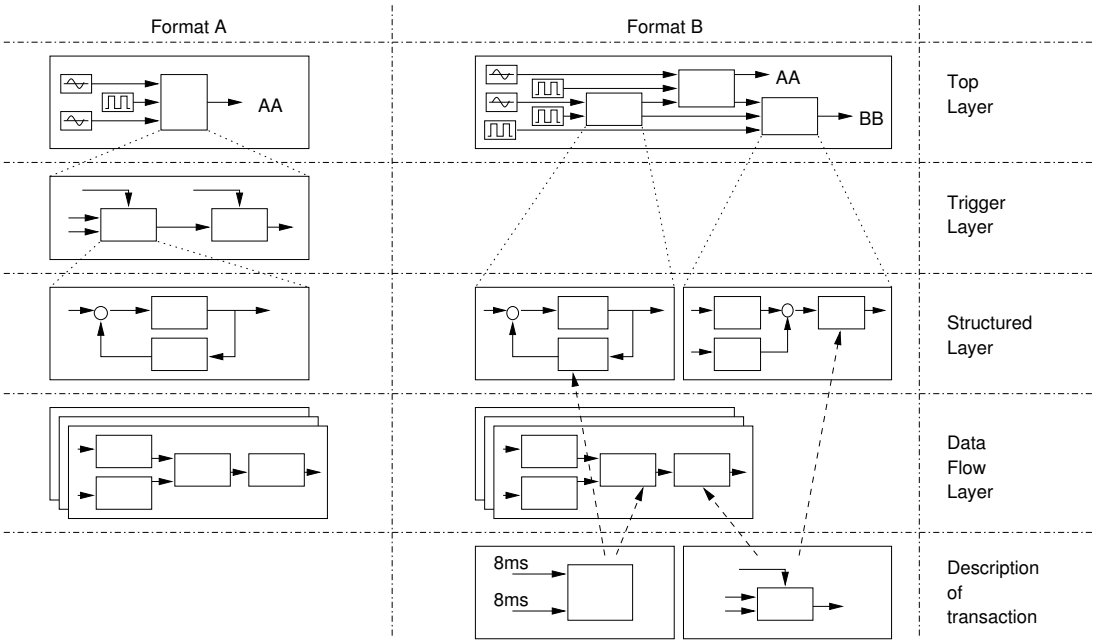

Figure 2: J-MAAB's layered architecture (adapted from [21])

subsystems of the top layer in case of *format B*). This layer is not limited to a single subsystem and can be more hierarchically structured by defining further subsystems.

- *Data flow layer*: This layer finally defines the control logic (data flow) of the functional parts by using standard blocks. Of course, further hierarchical structures are allowed as well. No design patterns are provided by this layer.

In addition, a so-called *procedure timing description* has to be defined in case the triggered layer is not defined (i.e. *format B* is used). This is done by the use of Unit Delay blocks, which have to be added to the structured and/or the data flow layer in order to model the execution of blocks at different rates as an *alternative way*. Unfortunately, the J-MAAB guideline provides no hints, neither which of the two formats should be applied to what application nor which format provides the most benefits in a general application case. Finally, rules about the initialization of the model (format independent) rounds out the architectural guidelines chapter.

Beside architectural guidelines, the document consists of further different sets of guidelines based on various topics. These guidelines touch the following topics: naming and format conventions, and methods of drawing signal lines and data types. Compared to the MAAB guidelines, however, only a minor number of guidelines are presented. Nevertheless, further topics like the design of data

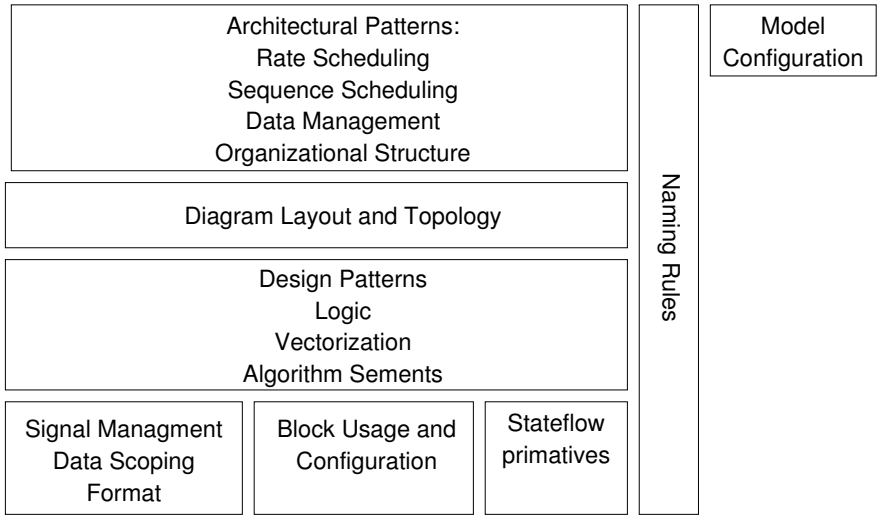

Figure 3: Guideline topics (adapted from the slides of the presentation held at the MAAB Style Guide for 2006 workshop)

dictionaries (partly available as an appendix to the J-MAAB guidelines) and a function guide are under consideration. Finally, a guide about convenient combinations of Simulink blocks (i.e. a modeling cookbook) is planned to be published separately to the Simulink StyleGuide document.

A preview to the MAAB Style Guide for 2006[14] Unfortunately, the MAAB guidelines are slightly outdated. The periodic updates mentioned in the document did not happen. In the meanwhile, in-house guidelines have been developed (e.g. by MAAB member companies) or alternative working groups like the J-MAAB community have been established. In addition, the NA-MAAB[15] (North America MAAB) and E-MAAB (Europe MAAB) working groups have been founded as regional sub-communities.

During the International Automotive Conference (IAC) 2006, MathWorks has foreshadowed the publication of a revised version of the MAAB guidelines in order to collect the experiences gained by the MAAB members, sub-communities (NA-MAAB, E-MAAB) and related communities (J-MAAB) over the last years. Figure 3 shows the planned or revised topics of the new guideline document. Major input regarding architectural patterns is expected to be done by the J-MAAB

[14]The information presented in this paragraph are gained at the *MAAB Style Guide for 2006 workshop* at the *International Automotive Conference* (IAC) 2006 in Stuttgart. The slides of the workshop are available by MathWorks upon request and contain a preview of the most important changes and rules as well.

[15]NA-MAAB has members from General Motors, DaimlerChrysler, Ford Motor, Delphi, Visteon, Freescale, Ricardo, L3 Titan Systems (Nasa related), Deere, CAT and Lear.

working group. Concerning the old MAAB guidelines, some rules will be dropped, updated or adapted in order to be applicable to the current tool technology. In addition, the guideline format will undergo slight modifications (e.g. the benefit and penalty issues are combined to a field named *rationale*; dropping the automation field, etc.). In the new document, the scope field has as possible entries either MAAB (if all regional communities agree) or NA-MAAB, J-MAAB, E-MAAB. The structure of the document will distinguish between general and naming convention rules as well as between Simulink and Stateflow guidelines. Subchapters for Simulink will be about further Simulink-specific general and block-specific rules, guidelines for diagram topology and layout and will also contain a list of Simulink patterns. The Stateflow chapter will consist of general rules as well as of guidelines for events and state machines. A list of Stateflow pattern will round out the proposed chapter. The new MAAB guidelines will be published in various formats including XML (for tool exchange) and HTML (see the description of the *eGuidelines tool* in the next section). It is planned to merge the materials of the regional working groups by 2006.

dSpace Guidelines for TargetLink (published in 2006) Code generators are an essential part of many modeling tools and liquidate the restriction of using them for simulation only. Although MathWorks has established its own code generator, namely the Real-Time Workshop add-on, TargetLink provided by dSpace is a widely established alternative to the MathWorks product. Since the MAAB guidelines mainly consists of a set of generic and application independent rules, dSpace decided to publish their own guidelines [12][16]. These are partly based on or are at least influenced by the MAAB guidelines, but extend them by three kinds of guidelines:

First, the guideline document defines a suitable language subset. In a very rudimentary form, such guidelines can be already found in the MAAB guideline document, for instance the prohibition of using continuous blocks in controllers. These rules are supplemented by further guidelines. For instance, the avoidance of algebraic loops is strongly recommended by the dSpace document. Another example is the proscription of using block priorities to influence the block update order (see Table 3) or restrictions of the use of a certain block like the Merge block (see Table 4) . Consequently, these kinds of guidelines have the same intention as the MISRA-C guidelines: avoiding common programming and design errors by using only a language subset that has been proved to be safe.

The second focus of the document is about rules regarding the generation of efficient code, especially with the use of TargetLink. TargetLink offers a huge list of modeling options in order to generate highly optimized code. The guideline document gives hints of how to use those options efficiently. In addition, modeling

[16] `http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/modeling_guidelines.cfm`

Topic	*Block Priorities*
Rule	Block priorities must not be used to specify the block execution order in Simulink
Purpose	Code generation with TargetLink
Remark	The use of block priorities leads to an intransparent modeling style and is therefore not supported (ignored) by TargetLink
References	• TargetLink Production Code Generation Guide • General Limitiations • Block priorities • Block scheduling
Example	see [12]

Table 3: A rule forbidding algebraic loops (adapted from [12]).

Topic	*Restrictions with Regard to the Merge Block*
Rule	The *Allow unequal port* widths parameter must not be checked for the Merge block.
Purpose	Code generation with TargetLink
Remark	TargetLink does not support the option for the reason of code efficiency.
References	• TargetLink Production Code Generation Guide • TargetLink Limitations • Block-Specific Limitations • Merge block
Example	see [12]

Table 4: A rule forbidding a certain block option (adapted from [12]).

patterns are presented with the intention of introducing a modeling style that can be converted to efficient code. For instance, design patterns for control flows in Stateflow are presented.

Finally, an extensive chapter of the dSpace document is dedicated to rules that explain how to generate code that applies to the MISRA-C standard or at least ensures that a maximum of MISRA-C rules are accomplished. For instance, the guidelines explain how the generation of recursive functions can be avoided.

Beside these three major issues, the document presents and updates an enhancement of the MAAB layout rules. Additionally, guidelines for comments, name spaces as well as model and simulation parameters are mentioned as well. Each guideline of the whole document has a simple structure and the descriptions are short and easy to understand. Furthermore, each guideline is motivated in a sense that the user understands the benefits of the rule as well as the consequences if the rule is violated. Compared to the MAAB guidelines, the dSpace guidelines are a reasonable endorsement and a good example of a more application-specific enhancement of the MAAB rules. By the time the current report is written, there has been, however, no direct cooperation with ongoing or current activities of the several MAAB communities. Since the dSpace guidelines are not TargetLink specific only, such a cooperation in the near future would be a further important step towards developing a universal guideline document.

MISRA Guidelines (announced to be published in fall 2006) The MISRA consortium[17] currently develops its own guidelines regarding model-based software development and generating MISRA-C compliant code by the use of automatic code generator tools [15]. Beside activities regarding development of standards for UML and modeling tool independent guidelines, modeling guidelines for Simulink and Stateflow (including the corresponding code generator add-ons) are currently developed as well. A participation to the NA-MAAB styleguide working group in order to provide an extension to MAAB-related guidelines is planned as well. At the time of this report, the work on the MISRA guidelines is still in progress. According to information from the MISRA consortium, the first documents are planned to be published in fall 2006.

Additional Stateflow Guidelines The model abstraction used by Stateflow is mainly based on Harel's Statecharts [19]. Compared to Simulink, the model abstraction is therefore founded on a widely discussed research topic. Consequently, it is easy to find discussions about the benefits and drawbacks of statecharts, which are often enclosed by modeling guidelines. Now, statecharts are a fixed part of the UML diagram collection. The book *The Elements of UML 2.0 Style* [2], contains style guides for UML diagrams including ones for statecharts. More guidelines and patterns, especially focusing on the automotive domain and

[17] `http://www.misra.org.uk`

designed for Stateflow, can be found in [7] and [31, 32]. The models that should adhere to the rules presented in the latter publications are verifiable by a prototype of a corresponding model checker tool.

2.3 Tool Support

In the previous section, we mentioned tool support for checking general statechart-related guidelines. This section presents several tools that are applicable to MAAB-related guidelines, however, are not limited to it. Furthermore, these tools and the guidelines they are delivered with constitute another source of rules and guidelines that goes beyond the generic MAAB guidelines.

MathWorks' ModelAdvisor The current release of MATLAB/Simulink[18] contains a tool named *ModelAdvisor*[19]. This tool is suited to check various model consistency, optimizations and performance issues. Consequently, the tool provides further guidelines and rules in order to improve the quality of Simulink models that can be added to already mentioned publicly available guidelines. Moreover, the tool possesses an open API which allows enhancements by further rules. In principle, it is thus possible to integrate the MAAB guidelines into this tool as long as a specific MAAB-related guideline is well-suited for automated verification. Unfortunately, no integration of the MAAB rules has been done so far.

Mint tool Ricardo offers a tool called *mint*[20], which is able to check rules and guidelines for a given Simulink and/or Stateflow model as well. In general, mint works independent form a specific set of guidelines. Nevertheless, mint directly supports the MAAB guidelines by offering a corresponding subset of rules. In addition, Ricardo is currently working on an own subset of guidelines, which, however, are not published so far. As a further service, in-house or project specific guidelines can be added to the tool either by using an open API or by engaging Ricardo to write these tests.

eGuidelines This tool offers a web-frontend in order to organize, publish and administrate guidelines as well as to support the process of model reviews [10]. Filters, a search engine as well as links between related rules ease the navigation through the various guidelines. In addition, a printed version of a specific guideline database is also supported by the tool.

[18] Release 2006a

[19] `http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/ug/f4-141979.html`

[20] An evaluation version is available at `http://www.ricardo.com/engineeringservices/controlelectronics.aspx?page=mint`

The eGuideline tool was developed in the context of the IMMOS[21] (Integrierte Methodik zur modellbasierten Steuergeräteentwicklung[22]) project [33], a joint research of automotive companies, tool developers and research institutes[23]. The purpose of this project is the compilation of an abstract description of all major entities (including their interrelationships), which are needed for model-based development.

Currently, the *eGuidelines* are used and evaluated by DaimlerChrysler Research and Technology (DCRT)[11]. DCRT provides a demo version of the tool as well.[24] This version has access to a database, which consists of a subset of the MAAB guidelines (which are considered to be most essential) and further, DCRT-related guidelines. Internally, DCRT evaluates actually about 200 guidelines. Finally, it is planned to establish the *eGuideline* tool as the standard front-end tool for both the revised MAAB guidelines and the currently developed MISRA guidelines [15].

Model verification tools Finally, several model verification tools for Simulink and Stateflow are available on the market. These tools use formal verification technologies in order to prove if a certain model adheres to given requirements, which are usually defined by the use of formal terms as well. For instance, the *SCADE* tool[25] from *Esterel Technologies* is able to transform certain Simulink/Stateflow models into Lustre code. Then a verification engine checks the code against the formulated properties. Additionally, restrictions regarding the used block set are needed in order to fulfill formal verification criteria. These restrictions, which have a formal underpinning, are usually subsumed by further guidelines. As a further feature, the SCADE tool is able to check if a certain model adheres to those guidelines.

In addition to the SCADE tool, a further verification tool is offered by MathWorks itself by the add-on *Simulink Verification and Validation*[26]. Finally, another established tool, the *EmbeddedValidator*[27], designed especially for Simulink, Stateflow and TargetLink, is provided by *OSC Embedded Systems*.

[21] `http://www.immos-projekt.de/`

[22] An adequate English translation might be: An integrated method for model-based development of control systems.

[23] Members of the IMMOS project are DaimlerChrysler, dSpace, Fraunhofer-Institut für Rechnerarchitektur und Softwaretechnik, Forschungszentrum Informatik an der Universität Karlsruhe, IT Power Consultants and the Universität Paderborn (Institut für Informatik).

[24] `http://www.eguidelines.de`

[25] `http://www.esterel-technologies.com/products/scade-suite`

[26] `http://www.mathworks.de/products/simverification`

[27] `http://www.osc-es.de/products/de/EV.php`

3 Conclusions

In this document, the necessity of the development of modeling guidelines is motivated especially for Simulink and Stateflow. For this reason, a list of public available modeling guidelines is presented. Quality aspects applied to the modeling guidelines are worked out by the document. In addition, where applicable, a categorization of the guidelines according to common software engineering terms is suggested. Finally, future and current trends in the development of guidelines are sketched.

A general conclusion of this report is that the amount of public available documents is surprisingly low. From 1999 until 2006 only a fistful guideline documents have been published. A strategy rethinking of many companies towards publishing their in-house guidelines would be promising. Positive consequences of such a revised strategy would support cost reduction by preventing the ineffective reinvention of the wheel. Additionally, since many experiences of former projects are incorporated, quality issues of the related software products could be increased significantly. A convincing motto is presented by the J-MAAB community: "Cooperate with tools and compete with products".

References

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[2] Scott W. Ambler. *The Elements of UML 2.0 Style.* Cambridge University Press, 2005.

[3] Michael Beine, Ulrich Eisemann, and Christian Wewetzer. Quality assurance aspects and activities in automotive model-based development. 2006.

[4] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: The Signal language and its semantics. *Sci. Comput. Program.*, 16:103–149, 1991.

[5] G. Berry. *The foundations of Esterel.* MIT Press, 2000.

[6] Christian D. Bodemann and Antonio De Luca. Function-oriented model based design development for real-time simulators with matlab/simulink. In *Proc. Model-based Design Conference (MBDC)*, pages 79–85. The MathWorks, 2005.

[7] Daniel Buck and Andreas Rau. On modelling guidelines: Flowchart patterns for stateflow. *Softwaretechnik- Trends*, 21(2), 2001.

[8] Mary Campione, Alison Huml, and Kathy Walrath. *The Java Tutorial: A Short Course on the Basics*. Addison Wesley Professional, 2000.

[9] L. P. Carloni, Alberto L. Sangiovanni-Vincentelli, C. Hylands, Edward A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Seeking equilibrium between communication and computation in system-level design. Overview of the Ptolemy project. Technical Report UCB/ERL M03/24, EECS Department, University of California, Berkeley, July 2003.

[10] Mirko Conrad, Heiko Dörr, Ines Fey, Hartmut Pohlheim, and Ingo Stürmer. Guidelines und Review in der modell-basierten Entwicklung von Steuergeräte-Software. In *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik*. Expert-Verlag, 2005.

[11] Mirko Conrad, Ines Fey, and Hartmut Pohlheim. eguidelines - a tool for managing modeling guidelines. In *Proc. International Automotive Conference (IAC)*, 2005.

[12] Ulrich Eisemann. Modeling guidelines for function development and production code generation. Technical report, dSpace, 2006.

[13] Tom Erkkinen. Model style guidelines for flight code generation. In *Proc. Modeling and Simulation Technologies Conference*, pages 1–8, August 2005.

[14] Ford Motor Company. *Modeling Style Guidelines for Structured Analysis and Design using Matlab/Simulink/Stateflow*, 1999.

[15] Simon Fürst, Dietmar Kant, and Mirko Conrad. Misra modelling guidelines - scope of german participants. Technical report.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] Eran Gery, David Harel, and Eldad Palachi. Rhapsody: A complete life-cycle model-based development system. In *IFM*, pages 1–10, 2002.

[18] N. Halbwachs, P. Caspi, P. Raymond, and Pilaud D. The synchronous dataflow programming language Lustre. *Proc. of the IEEE, 79(9)*, 1991.

[19] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[20] IEC. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*, 1998.

[21] Japan MathWorks Automotive Advisory Board (J-MAAB). *Simulink StyleGuides*, March 2003.

[22] Gary W. Johnson and Richard Jennings. *LabVIEW Graphical Programming.* McGraw-Hill Professional, 2006.

[23] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C.* Carl Hanser Verlag München Wien; Prentice-Hall International Inc., London, second edition, 1990.

[24] Christoph M. Kirsch. Principles of Real-Time Programming. *LNCS*, 2491, 2002.

[25] Jeff Langr. *Essential Java Style.* Prentice Hall, 2000.

[26] The MathWorks. *Controller Style Guidelines for Production Intent Using MATLAB®, Simulink® and Stateflow®*, April 2001.

[27] MathWorks, www.mathworks.com. *Stateflow User's Guide*, March 2006.

[28] MathWorks, www.mathworks.com. *Using Simulink*, March 2006.

[29] The Motor Industry Software Reliability Association (MISRA). Guidelines for the Use of the C Language in Critical Systems. Technical report, MISRA, Oktober 2004.

[30] MSR-MEGMA. *Standardization of library blocks for graphical model exchange*, 2001.

[31] Martin Mutz. Metriken für zustandsbasierte software-entwicklung. *Metrics News*, 9, December 2004.

[32] Martin Mutz and Michaela Huhn. Automated statechart analysis for user-defined design rules. Technical report, 2003.

[33] H. Schlingloff, C. Sühl, H. Dörr, M. Conrad, J. Stroop, S. Sadeghipour, M. Kühl, F. Rammig, and G. Engels. Immos - eine integrierte methodik zur modellbasierten steuergeräteentwicklung, July 2004.

[34] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.

[35] Gerald Stieglbauer and Andreas Werner. Modellierung von deterministischer Software in Simulink. *Inform., Forsch. Entwickl.*, 19(4):189–193, 2005.

[36] Josef Templ. TDL Specification and Report. Technical report, Department of Computer Science, University of Salzburg, Austria, http://www.SoftwareResearch.net/site/publications/C059.pdf, March 2004.

[37] Artisan Software Tools. Artisan Studio Support for Model Driven Architecture (MDA). Technical report, 2002.

[38] Achim Wohnhaas, Rainer Moser, and Peter Brangs. Standardisierte Blockbibliothek zum modellbasierten Steuergeräte-Softwareentwurf als Basis für den Modellaustausch zwischen Entwicklungstools. Technical report, MSR-MEGMA, 1999.