Cornerstone - counter-argument to the optimism of your first two hypothese.
powerful evidence for your third hypothesis

JOT JOURNAL OF OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

The Language of SysML v2 under the Magnifying Glass

Nico Jansen*, Jerome Pfeiffer†, Bernhard Rumpe*, David Schmalzing*, and Andreas Wortmann†

*RWTH Aachen University, Germany

†University of Stuttgart, Germany

ABSTRACT The Systems Modeling Language (SysML) is defined as an extension of UML that reuses, forfeits, and adjusts selected parts of UML to facilitate the modeling of complex systems. While it has been used in a variety of domains to successfully design, analyze, develop, construct, and maintain such systems, its expressiveness is limited by its foundations in UML. SysML v2 will succeed its predecessor without being backward-compatible. It, thus, is designed from scratch as a textual language with model-based systems engineering in mind, giving the consortium working on its standardization the freedom to employ state-of-the-art language engineering guidelines, methods, and tools to improve the language. We examine how much SysML v2 adheres to language engineering guidelines and best practices by collecting such practices, identifying those relevant to SysML v2, and applying them to the language. Overall, we find that SysML v2 fulfills many guidelines regarding its functional suitability and usability but might pose challenges regarding its maintainability and portability.

KEYWORDS Model-based Systems Engineering, SysML, Language Engineering

1. Introduction

Systems engineering (Blanchard et al. 1990; Kossiakoff et al. 2011) is the ever-growing challenge of designing and developing complex systems, considering sociotechnical needs, an ever-growing body of technologies, evolving tools, and a diverse workforce. Systems engineering consist of various activities (Buede & Miller 2016), from gathering requirements to system design and development to test, evolution, and maintenance. A major hindrance in systems engineering is the abundance of natural language documents describing different aspects of the systems under development. These can hardly be processed automatically, their ambiguity raises confusion and misunderstandings, and their translation into actionable artifacts is tedious and error-prone. Model-based systems engineering (Wymore 2018) aims to address these challenges by leveraging models with various roles to abstract from concrete realizations. Yet, models in systems engineering often are less precise, informal, descriptions that are not based on formally defined modeling languages (i.e., sketches). While sketching is a valuable technique to foster thinking and communicating about designs (Bucchiarone et al. 2021; JSD+20 2020), translating sketches into (executable) artifacts rarely is possible due to their incomplete nature. The efficient use of models, e.g., for automated analyses, code generation, continuous integration, and deployment, demands well-designed modeling languages that not only foster understanding but also support their automated processing. A prominent example of a modeling language for developing cyber-physical systems is the Systems Modeling Language (SysML), a family of different languages to support systems engineering. Engineers already use SysML to develop large-scale systems in various domains (Bone & Cloutier 2010), including automotive, avionics, and medicine.

SysML v2 is being developed as the next-generation systems modeling language, intended to improve the precision, expressiveness, interoperability, consistency, and integration of the language concepts relative to SysML v1 (Group 2017). Language engineering is challenging as it demands understanding the subject of interest and carefully crafting abstract representations thereof, creating implementations capturing the language's syntax and semantics, integrating these properly, and providing appropriate tool support. To guide language engineers in developing modeling and domain-specific languages (DSLs), the

JOT reference format:
Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. *The Language of SysML v2 under the Magnifying Glass*. Journal of Object Technology. Vol. 21, No. 3, 2022. http://dx.doi.org/10.5381/jot.2022.21.3.a11

modeling and language engineering community has proposed a substantial corpus of best practices (Czech et al. 2018). We apply the proposed guidelines to the reference implementation of SysML v2 in Xtext (Bettini 2016). Due to the standardization of SysML v2 being driven by a large consortium of academic and industrial partners, we cannot expect large changes to the language anymore. Moreover, the reference implementation is already being used to develop tooling for SysML v2, such as its model API, Jupyter notebooks, and editors. Our investigations serve to highlight potential improvements to SysML v2 and to investigate the applicability of the guidelines raised by the software language engineering (SLE) community to one of the largest industrial SLE efforts to date.

The contribution of this paper, thus, is an analysis of the current status of the SysML v2 reference implementation with respect to proposed guidelines and best practices for SLE. We discuss the SysML v2 in the form of its most up-to-date "2022-01"[1] release and its compliance to these. While first releasing a textual syntax only, the developers of the SysML v2 have since released a graphical syntax as well. However, we here discuss the SysML v2 considering the textual syntax only, which had the opportunity to mature due to its earlier release. Due to its size, we cannot provide the SysML v2 grammar but instead refer to the latest official release candidate. Also, we will not compare SysML v1 and v2 in this paper. While being a relevant investigation, our contribution focuses on the language engineering aspect of SysML v2.

In the remainder, Section 2 gives background on modeling languages, SysML, and language design guidelines. Section 3 discusses the application of the language design guidelines to SysML v2 in detail before Section 4 recapitulates our findings and Section 5 suggests improvements. Section 6 then discusses the methodology of our analysis of SysML v2, and Section 7 considers related work. Finally, Section 8 concludes.

2. Background

2.1. Modeling Languages

To make models machine-processable, these must adhere to systematic descriptions–modeling languages. A (modeling) language is defined by a set of sentences (models) it comprises, it can be defined through (GR11 2011) (1) an abstract syntax, which structures its sentences; (2) a concrete syntax, which defines the appearance of its sentences; (3) a well-understood semantic domain, which enables giving meaning (Harel & Rumpe 2004) to its syntax; and (4) a semantic mapping from its abstract syntax to the semantic domain, which assigns meaning to models.

Modeling language implementations often use grammars (Bettini 2016; HRW18 2018), metamodels (Steinberg et al. 2008; Degueule et al. 2015), or projectional editing (Campagne 2014) to define the syntax, as well as model-to-model (Jouault et al. 2006; HRRW17 2017) or model-to-text (Forsythe 2013; Bettini 2016) transformations to realize their semantics. Most mechanisms used for defining abstract syntaxes are context-free, i.e., they cannot express contextual properties (e.g., the name of a model element is unique in its scope). Consequently, many language implementations feature additional well-formedness rules (e.g., OCL, Java, or similar) to further constrain the language's models. Sometimes, this is called "static semantics" (CBCR15 2015).

[1] Available from https://github.com/Systems-Modeling/SysML-v2-Release/

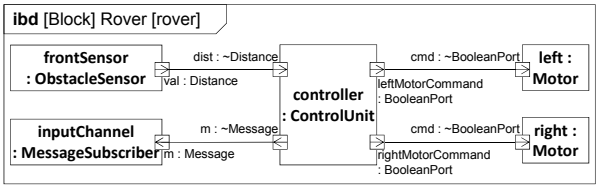

Figure 1 Overview of a rover robot in SysML v1

2.2. SysML v1

The SysML v1 is a graphical, general-purpose modeling language family based on UML (OMG 2011) that comprises nine sub-languages. But where UML targets the engineering of software systems, SysML aims to facilitate the engineering of cyber-physical systems through specification, analysis, design, validation, and verification. To this end, SysML adopts six of UML's diagrams, introduces changes to these, and introduces the block definition diagram, internal block diagram, and parametrics diagram. Block definition diagrams correspond largely to class diagrams with ports and are used to define data types as well as to define component types. Internal block diagrams describe the configuration of block definitions. Parametrics describe constraints between SysML elements. Overall, the pillars of SysML are (1) Structure: block definition diagrams, internal block diagrams; (2) Behavior: sequence diagrams, state machine diagrams, activity diagrams; (3) Requirements: requirements diagrams; and (4) Parametrics: parametric diagrams.

SysML has been applied to the engineering of a variety of complex systems, including buildings (Cawasji & Baras 2018), cars (Andrianarison & Piques 2010), the thirty meter telescope (Herzig et al. 2017), satellites (Hayden & Jeffries 2012), and many more, despite lacking formal semantics (in the sense of "meaning" (Harel & Rumpe 2004; Hamilton & Hackler 2007; Lima et al. 2013). Lacking a formal specification of the meaning of SysML elements prevents exchanging its models between different companies and tool vendors, which need to fill the semantic gaps themselves and, thus, hide the meaning of language elements in their (most often) proprietary and vendor-locked tool implementations. Moreover, this often produces specific realizations of semantics that can be incompatible with another, similar to the different interpretations of UML as manifested in its variants (such as fUML (OMG 2021), txtUML (Dévai et al. 2014), or UML/P (Rum16 2016)). This can lead to confusion and misunderstandings to erroneous analysis and processing of models. Moreover, modeling languages with formal semantics (e.g., based on CSP (Hoare 1978), the π calculus (Sangiorgi & Walker 2003), or the FOCUS theory (Broy & Stølen 2001)) support richer analyses and semantics-aware development methods (KRW20 2020; BKRW19 2019).

Figure 1 illustrates the structural pillar of SysML with an excerpt of an internal block diagram (ibd) describing the architec-

ture of an autonomous rover (DMW17 2017) that receives inputs from two sensor blocks (`frontSensor` and `inputChannel`), decides upon the next action in its `controller` block, and actuates two parallel blocks of type `Motor` accordingly. The architecture exchanges messages between blocks using connectors between their typed and directed ports. Its blocks may be composed of further blocks.

2.3. SysML v2

SysML v2 is the successor of SysML and currently in definition. It is defined as a textual modeling language[2] that refines and harmonizes the concepts of its predecessor and improves its syntax. SysML v2 also forsakes the distinction into different model (diagram) types, defines an API for its models to ease exchanging these, and comes with a pilot implementation comprising an Xtext (Bettini 2016) grammar. The new syntax, moreover, features variation, snapshots, quantities and units, complex statements and expressions, and a library of types and functions to operate on.

SysML v2 is defined as a language extension of KernelML[3], a new modeling language foundation part of the SysML v2 standardization, which defines essential concepts that could be reused for creating SysML variants or other similar languages. These concepts include packages, classifiers, associations, connectors, expressions, and similar foundations.

Figure 2 illustrates the textual syntax of SysML v2 on the rover robot of Figure 1. It features a central package that contains various type definitions of parts, ports, and interfaces. The part definition of the `Rover` (ll. 19 f.) instantiates various parts and connects these accordingly.

2.4. Language Design Guidelines

Starting with a recent systematic mapping study (Czech et al. 2018) and selected publications (Selic 2009) on guidelines for SLE (Karsai et al. 2009; Zaytsev 2014; de Kinderen 2017; Petrausch et al. 2016), we applied forward and backward snowballing to identify further sources of SLE guidelines. From the publications, we extracted all guidelines and removed those that are hardly falsifiable ("care for quality" (Kolovos et al. 2006)), relate to the SLE process in a way such that they cannot be checked post-hoc ("Effective process for DSML definition" (Kahlaoui et al. 2008)), cannot be applied to SysML v2 ("Use colors" (Kelly & Tolvanen 2008)), or relate to tooling for the modeling language ("Filter details from notational elements" (Kelly & Tolvanen 2008)).

The remaining guidelines are presented in the following and cover different areas of ISO 25010 (ISO/IEC 2010) (cf. Figure 3) for software product quality. Software languages succumb to the same quality and maintenance issues as other software systems, raising the importance of complying with high standards in development and evolution. Following this classification scheme, we analyze the current version of SysML v2. Since the guidelines are sometimes underspecified and miss preciseness,

[2] A graphical notation to complement the textual notatoin is in progress – see "Intro to the SysML v2 Language-Graphical Notation.pdf"

[3] See "Kernel Modeling Language (KerML)", Version 1.0, Release 2022-01.

```
01 package 'EXE 2017 Rover' {
02   import ScalarValues::*;
03
04   // type definitions
05   part def Motor             { ... }
06   part def ControllerUnit    { ... }
07   part def ObstacleSensor    { ... }
08   part def MessageSubscriber { ... }
09
10   port def BooleanPort {
11     out signal        : Boolean;
12   } // additional port definitions
13
14   interface def MotorCommand {
15     end src           : BooleanPort;
16     end tgt           : BooleanPort;
17   } // additional interface definitions
18
19   part def Rover {
20     // instantiation of parts
21     part left         : Motor;
22     part right        : Motor;
23     part frontSensor  : ObstacleSensor;
24     part inputChannel : MessageSubscriber;
25     part controller   : ControllerUnit;
26   }
27
28   part rover : Rover {
29     interface : MotorCommand connect
30       src => controller::leftMotorCommand to
31       tgt => left::cmd;
32       // additional interface usages
33   }
34 }
```

Figure 2 Excerpt of the rover robot of Figure 1 in SysML v2

we use our expertise in the domain to interpret them and evaluate SysML's compliance. In the following, we organize the guidelines found in literature according to the taxonomy of ISO 25010 qualities as illustrated in Figure 3 into guidelines regarding functional suitability, usability, maintainability, portability, and compatibility.

2.4.1. Functional Suitability Functional Suitability describes the ability of a software system to offer necessary or beneficial functionalities that support end-users in performing their tasks. For modeling languages, this includes the capability to completely and correctly create models with respect to a particular domain.

G1: Balance generality and specificity (Karsai et al. 2009; Völter 2009; Wile 2004). Modeling languages should support abstracting from implementation details. At the same time, they must offer expressive modeling techniques. While these requirements often contradict each other, finding an appropriate equilibrium is crucial.

G2: Always start with a semantics model (Selic 2009). The purpose of modeling language is to express something with suitable abstraction, i.e., to carry meaning. This meaning, the language's semantics (Harel & Rumpe 2004) need to be established before any syntax is defined. If the syntax is defined first, its element might not be suitable to carry the semantics properly and it raises the danger of yielding a 'modeling language' without meaning (cf. UML, SysML).

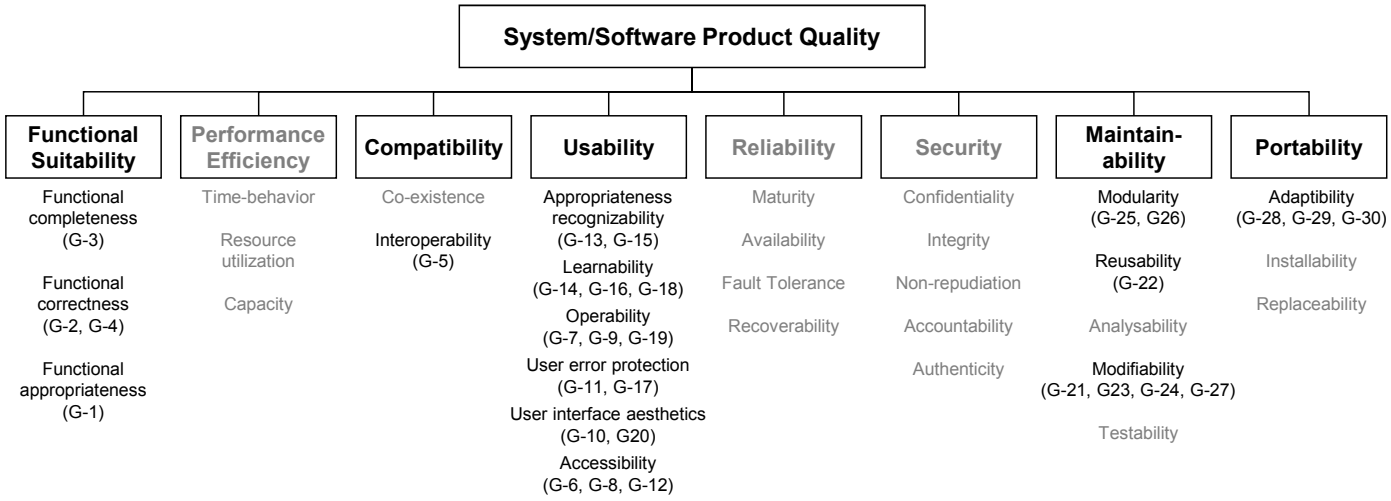

Figure 3 The software product quality model defined in ISO/IEC 25010 comprises eight quality characteristics.

G3: Defined scope and purpose (Jannaber et al. 2017). When developing a DSL, the language designer has to decide on how specific and whether or not to develop it for a particular domain. To define the scope of the language, its purpose has to be evaluated, e.g., in respect to future modeling projects. Without a defined scope and purpose, a DSL is either overly general or too complex and specific for the targeted modeling subject.

G4: The modeling language is specified by a language metamodel (Jannaber et al. 2017). Specifying the modeling language in a language metamodel includes defining the abstract syntax, concrete syntax, well-formedness rules, and semantics in a metamodeling language. This language metamodel facilitates easy understanding of the scope and elements of the language and provides a standardized way for further changes and adaptions. Consequently, missing metamodels impedes the development, use, extension, and reuse of DSLs.

2.4.2. Compatibility

Compatibility describes the ability of a software system to be adapted or ported to another application area. This is an essential aspect for software languages as they can be integrated, composed, or adapted to other languages.

G5: Provide integrability (Kolovos et al. 2006; Kelly & Tolvanen 2008). Sophisticated software systems integrate well with other applications. This is also essential for modern modeling languages to be embedded in tools or composed/coordinated with other languages. Thus, they provide genuine added value across different application domains.

2.4.3. Usability

Usability is the capacity of a modeling language to provide the conditions for its intended users to perform tasks safely, effectively, and efficiently in a satisfactory fashion.

G6: Adopt existing domain notations (Karsai et al. 2009; Kelly & Tolvanen 2008; Völter 2009; Wile 2004; Mernik et al. 2005; Frank 2013). To provide a suitable language, the concrete syntax should adhere to concepts known by the modelers. Generally, these originate from associated domains, providing an intuitive understanding of notations and reducing the entry barrier.

G7: Avoid redundancy (Karsai et al. 2009; Kolovos et al. 2006; Salehi et al. 2016). The concrete syntax of a language defines valid sentences for modeling. To increase the comprehensibility of models, it is useful to establish a unique mapping between the syntactic elements and the underlying concepts to be represented. For this reason, redundant syntactic constructs should be avoided.

G8: Viewpoint orientation (Kahlaoui et al. 2008; Kelly & Tolvanen 2008; Völter 2009). Views enable to examine models with respect to specific concerns or to specify valid or invalid, as well as dependent or alternate designs. Through view support in modeling languages, modelers can examine portions of an area of interest, potentially mitigating the comprehension effort of large-scale models.

G9: Make elements distinguishable (Karsai et al. 2009; Kelly & Tolvanen 2008; Kelly & Pohjonen 2009). Software languages offer a variety of different modeling techniques. For the convenience of modelers, the provided syntactic elements should be sufficiently distinct from each other. This reduces confusion and thus increases productivity.

G10: Consistent style everywhere (Karsai et al. 2009; Kelly & Tolvanen 2008). There are different styles for representing concepts syntactically. To make a language more accessible, the style should be chosen consistently (e.g., same notation of names or types everywhere). While there is overlap on the distinguishability of elements, this guideline is no contradiction to **G9**, as it relates to the general look and feel of a language.

G11: Define language rules (Kelly & Tolvanen 2008). Models are often subject to various rules derived from the domain, the language itself, or some usage conventions. They may be strict by informing about missing elements or incorrect usage and even support conventions and default values. Language rules or well-formedness rules enable to detect errors early and prevent

invalid or unwanted models.

G12: First things first - the language (Kelly & Tolvanen 2008). Modeling tools consist of several components, including the modeling language in question, code generators, supporting processes, and tools. However, while code generators and language tools are mostly invisible to modelers, the modeling language is the means by which they develop systems. Changes to a modeling language may have major impacts on modelers and their creations. Thus, the modeling language should be crafted with care early on the avoid late changes.

G13: Derive concepts from expected output (Kelly & Tolvanen 2008). While being complex and mostly providing general solutions, software languages are often well-structured and utilize well-thought concepts. Through abstraction, the target code can be the source of modeling concepts. While reusing concepts from the target domain promises an increase in productivity and quality, this should be done with care as it is easy to convolute already established concepts.

G14: Multiple level of abstractions (Frank 2013). Modeling languages may address various concerns, such as the behavior of its system or its structure. A modeling language should provide concepts to distinguish these concerns clearly. Overloading models with different concerns may compromise understanding and prevent an appropriate interpretation.

G15: Clear language to target mapping (Frank 2013). A mapping from a modeling language to the target domain, also called semantic mapping, provides the modeling language with meaning. Without a clear meaning, the modeling language does not have a purpose, is susceptible to ambiguous interpretations, and does not allow for formal analysis.

G16: Give the user flexibility in how to formulate a model (Walter & Masuch 2011). Many roads lead to Rome. Consequently, different modelers might use dramatically different approaches to reach the same or similar modeling goals (cf. flattened vs. hierarchical states in UML Statecharts). Yet, guiding the way how to express concepts is part of the motivation of modeling languages. Thus, when designing such languages, weighing the freedom of expression vs. the guidance supported by restrictions in syntax and semantics is essential.

G17: DSL's support for error prevention and model checking (Kahraman & Bilgen 2015). Model-checking and error prevention play an important part in producing reliable programs. Often this can be improved because the inspection of all relevant parts of a model for errors and completeness are either missing or incomplete. Consequently, when designing a DSL, providing model checking and error prevention measurements help modelers to design solutions to their problems.

G18: Comprehensibility and Learnability (Kahraman & Bilgen 2015). For a DSL to be comprehensible, language elements have to be understandable, e.g., by reading their description or doing a tutorial. With this, it is easier to learn the DSL and to develop programs with it. Overly complex DSLs without documentation are hard to understand and have a steep learning curve that drives users away.

G19: The language provides mechanisms for compactness of the representation of the program (Kahraman & Bilgen 2015). Where models tend to get large and complex, mechanisms for compactness in the representation of language elements can mitigate this effect. Therefore, it is essential to introduce concepts for compactness in your DSL. Compactness increases the development speed of modelers through shortcuts and also the usability by smaller and more accessible models.

G20: Provide a graphical notation (Jannaber et al. 2017). A graphical notation for a DSL increases its usability and comprehensibility. The definition of the notation represents the last building block of the language definition. Each language element needs to be assigned to a graphical notation.

2.4.4. Maintainability

Maintainability characterizes the effort required to make specified (planned) modifications to a modeling language through its lifecycle.

G21: Reuse and compose existing language definitions (Karsai et al. 2009; Völter 2009; Mernik et al. 2005; Kelly & Pohjonen 2009). Modeling languages often reuse existing concepts of other languages. To reduce the implementation and maintenance effort, it makes sense to reuse existing language definitions on the technical level as well (BEH+20 2020). Composing multiple fragments is a powerful technique to develop new languages sustainably.

G22: Support reusability of language constructs (Karsai et al. 2009; Kelly & Tolvanen 2008; Kelly & Pohjonen 2009). Typically, languages support common notations to provide consistency. To avoid redundant definitions, identical constructs should be reused (Combemale et al. 2018). This increases the maintainability of the language and facilitates its evolution.

G23: Design for language evolution (Völter 2009; Kelly & Pohjonen 2009; Livengood 2012; Vierhauser et al. 2015). As languages will inevitably evolve, they should be designed accordingly. This includes the modularization of their concepts. Considering evolution in language design allows modifying dedicated aspects of a language only, rather than causing profound changes.

G24: Reuse type systems (Karsai et al. 2009; Mernik et al. 2005). Type systems formalize and enforce data structure rules to avoid type errors of unsupported operations. The design of a well-thought type system must consider complex correlations such as type conventions, generic types, and polymorphism (BEH+20 2020). Employing and extending well-thought type systems improves comprehensibility and avoids errors through misunderstanding and implementation errors.

G25: Interface concept (Karsai et al. 2009). Interfaces between parts facilitate modular system development and separation of concerns by providing means for information hiding, thus enabling developers to exchange parts without affecting other elements of the system. Providing a concept for interfaces eases development and supports modularization.

G26: Modularize and layer the language (Selic 2009). Where modeling languages aim to describe complex systems, these often demand for multiple viewpoints on the system (such as structure, behavior, or requirements in SysML). Properly modularizing and layering the language can support using different syntaxes for the different viewpoints and decoupled evolution of language parts relating to these viewpoints (HMSNRW16 2016). To avoid fragmentation, all language parts should be based on (layers of) shared concepts.

G27: Modifiability (Kahraman & Bilgen 2015). Modifiability describes the amount of effort required to extend or alter the functionality of a DSL. The DSL should be designed such that modifying the DSL does not degrade its existing functionality. Modifiability is important to incorporate future requirements which minimal effort. If a DSL is not intended for modifiability, the process of incorporating changes is hard and expensive, so language designers may tend to re-development instead of reusing and adapting the existing DSL.

2.4.5. Portability

Portability is the capability of ability to transfer a modeling language into different contexts and application environments.

G28: Support variability on language level (Kelly & Pohjonen 2009). Sophisticated modeling languages should be designed for extension, harnessing advanced mechanisms of the language workbench they are developed with. Variability for languages (Butting et al. 2018) is an essential part of sustainable modeling, allowing domain-specific solutions for particular application areas.

G29: Provide for language extensibility (Selic 2009). Software languages are software too (CFJ+16 2016) and, hence, often subject to evolution beyond the conceptions leading to its first release(s). This especially holds where languages are relatively generic and will be specialized by future users, such as the UML with MechatronicUML (Burmester et al. 2004) or UML/P (Rum16 2016). Without language suitable extension mechanisms (cf. (HRW18 2018)), users will ultimately abandon such a language and create their own variants from scratch.

G30: Allow for incorporating foreign language fragments in models (Selic 2009). Often, languages do not require comprehensive extensions but cannot support a few specific concepts required by the modelers. Instead of reinventing the complete language, providing suitable extension points to embed fragments of other languages (e.g., SQL into Java or Statecharts into SysML block diagrams) can ease reusing the language and its tooling (BRW16a 2016). Similar to **G30**, lacking such extensibility will drive users away.

3. Analysis of SysML v2

Functional Suitability

G1: Balance generality and specificity (Karsai et al. 2009; Völter 2009; Wile 2004). Although it is always challenging, finding the correct level of abstraction is particularly hard for SysML v2 as it is an interdisciplinary modeling language. Domain experts require different views on a subject with distinct granularity. Therefore, SysML v2 provides different levels of abstraction in various ways. First, it is distinguished between definition and usage of an entity, allowing a more abstract view on definitions, which is detailed for the particular usage. Furthermore, SysML v2 supports decomposing model elements making it possible to always provide the correct level of granularity. Thus, while still in the responsibility of the modeler, SysML v2 facilitates balancing generality and specificity.

G2: Always start with a semantics model (Selic 2009). There are different conceptions about the semantics of a modeling language in the literature. Often, it is described as the behavior or well-formedness of a modeling language, where it should be its "meaning" (Harel & Rumpe 2004), usually given by mapping the languages' constructs to a well-known semantic domain (Cengarle et al. 2009) (e.g., Petri-nets for UML activity diagrams). For SysML v2, a definition of semantics in the sense of meaning is not available.

G3: Defined scope and purpose (Jannaber et al. 2017). Similar to UML or its predecessor, SysML v2 is a very general language. Hence, it is designed to support various application domains. However, its purpose is to facilitate model-based systems engineering and to support its specific methods and practices.

G4: The modeling language is specified by a language metamodel (Jannaber et al. 2017). The language metamodel of SysML v2 is incomplete. In its current state, the reference implementation contains an Xtext grammar[4] that defines the abstract and concrete syntax of the SysML v2. However, there are, besides referential integrity conditions, no well-formedness rules. An implementation for semantics is also missing.

Compatibility

G5: Provide integrability (Kolovos et al. 2006; Kelly & Tolvanen 2008). As the underlying language workbench Xtext (Bettini 2016) supports language integration to a certain extent, the integrability of the language is generally possible. Furthermore, there is a standardized API[5] for SysML v2 models, allowing further processing in external tools. Thus, SysML v2 supports this guideline on the meta and model level. Integrability in the sense of coordination among the SysML v2 languages and with other languages can be principally supported through the API of SysML v2 models.

Usability

G6: Adopt existing domain notations (Karsai et al. 2009; Kelly & Tolvanen 2008; Völter 2009; Wile 2004; Mernik et al. 2005; Frank 2013). As SysML v2 acts across domains, by design, its notation must serve all disciplines appropriately.

[4] https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation/blob/master/org.omg.sysml.xtext/src/org/omg/sysml/xtext/SysML.xtext

[5] https://github.com/Systems-Modeling/SysML-v2-API-Services

```
01 // G-7: Three equivalent connector alternatives
02 interface : MotorCommand connect
03   src => controller::leftMotorCommand to
04   tgt => left::cmd;
```

```
05 interface : MotorCommand connect
06   controller::leftMotorCommand to left::cmd;
```

```
07 connect controller::leftMotorCommand to left::cmd;
```

Figure 4 Redundant connector alternatives in SysML v2

However, since it largely adapts and extends existing modeling techniques (especially structure and behavior), it can be argued that this guideline is respected. Furthermore, SysML v2 can also be seen as a textual enhancement of the graphical v1, translating and improving existing notation textually. Thus, while altering graphical elements into primarily textual ones, SysML v2 adopts the main concepts such as blocks (i.e., part defs), parts, packages, ports, activities, states, transitions, etc.

G7: Avoid redundancy (Karsai et al. 2009; Kolovos et al. 2006; Salehi et al. 2016). SysML v2 introduces lots of abbreviations on model level (e.g., :> as a short form for specialization on type level and subsets on instance level at the same time). While those abbreviations arguably increase the productivity of more experienced modelers, it poses a barrier for novice users. Furthermore, there are multiple syntactical notations for the same language concepts. Figure 4 presents three notations for linking two ports of different parts. The first alternative (ll. 2-4) provides all information about an instance of the connection, while the second snippet (ll.5-6) omits the endpoints. The last example (l. 7) does not give any information on the connection type, thus resulting in an instance of a generally provided connection definition. While these alternatives syntactically offer different levels of detail, they nevertheless result in the same connection for the specified ports.

G8: Viewpoint orientation (Kahlaoui et al. 2008; Kelly & Tolvanen 2008; Völter 2009). The definition of SysML v2 comes with a mechanism for specifying custom views and viewpoints. A viewpoint characterizes portions of an area of interest. A view may adhere to several viewpoints to extract and render relevant model information. Consequently, SysML v2 enables the partial examination of a system and thus satisfies **G8**.

G9: Make elements distinquishable (Karsai et al. 2009; Kelly & Tolvanen 2008; Kelly & Pohjonen 2009). While not strictly differenced into four pillars as in the first version, the SysML v2 cleanly distinguishes elements by providing different kinds of modeling techniques (e.g., for structure, behavior, etc.). Additionally, SysML v2 follows a rigorous concept of determining definitions and usage, resulting in clearly distinguishable abstraction levels (see Figure 5).

G10: Consistent style everywhere (Karsai et al. 2009; Kelly & Tolvanen 2008). Although different concepts are suitably syntactically differentiated, SysML v2 also emphasizes a consistent notation of similar concepts. This is apparent, for example, in a uniform notation of types and instances (see Figure 5) or

```
01 // G-9 : Distinguish type definition and instance
02 // G-10: Consistent definition of and using types
03 part def Car {
04   part eng : Engine;
05   item fuel : Fuel;
06   attribute status : CarStatus;
07 }
08 part def Engine;
09 item def Fuel;
10 attribute def CarStatus {
11    gearSetting : Integer;
12    //...
13 }
```

Figure 5 SysML v2 clearly distinguishes type definitions and their instantiation. This style is consistent across the modeling elements, e.g., parts, attributes, and items.

```
01 // G-11: Missing language rules
02 package Robot {
03   import Rover::*;
04
05   part def Bot {
06     part left : Motor;
07     part right : Motor;
08     part controller : ControllerUnit;
09   }
10
11   part bot : Bot {
12     // connects a part to a port
13     interface : MotorCommand connect
14       controller to left:cmd;
15   }
16 }
```

Figure 6 A SysML v2 model showing questionable interface usage where a part is connected to a port.

common keywords concerning the extensibility of elements.

G11: Define language rules (Kelly & Tolvanen 2008). In the current version of SysML v2, there are almost no well-formedness rules. What constitutes a well-defined model is, therefore, completely open or guesswork. For instance, the compatibility of types is not checked or not required, as presented in Figure 6 for the connection of a part to a port (l. 14). However, well-formedness rules are not completely missing, so it is recognized when type definitions are not discoverable during usage. An extended set of well-formedness rules would nevertheless provide more clarity and make the set of valid models known.

G12: First things first - the language (Kelly & Tolvanen 2008). The main focus in the development of SysML v2 was apparently on developing the concrete syntax of the modeling language. Code generators and frameworks based on the language are not implemented yet.

G13: Derive concepts from expected output (Kelly & Tolvanen 2008). Besides their meaning, the purpose of SysML v2 models is also unclear. They are only of limited use for specification due to their ambiguity in interpretation. Moreover, they do not support formal analysis or comprehensive well-formedness rules. Thus, no concepts can be derived from the expected output of SysML v2 models, as there is no expected output.

```
01 // G-19: Detailed and short form of a binding
02 port def waterTankPort {
03   out item waterSupply;
04 }
05
06 bind waterTankPort::waterSupply = pump:pumpOut;
07
08 part pump : CoolingWaterPump {
09   out item pumpOut : Water;
10 }
```

```
01 port def waterTankPort {
02   out item waterSupply;
03 }
04 part pump : CoolingWaterPump {
05   out item pumpOut : Water
06                    = waterTankPort::waterSupply;
07 }
```

Figure 7 Two SysML models showing the definition of a binding connection (top, l. 6) and its shorthand version that combines the binding and feature definition (bottom, ll. 5-6).

G14: Multiple levels of abstraction (Frank 2013). SysML v2 supports modeling various concerns at different levels of abstraction. Even for elements of behavior modeling, such as actions or statecharts, a distinction is made between definition and usage. Whether to model everything in the same model or to separate multiple levels of abstractions is the decision of the modeler.

G15: Clear language to target mapping (Frank 2013). Although the SysML v2 operates in the broader spectrum of systems modeling, there is no clear mapping from the language to the domain of systems engineering. Whether elements represent software or hardware, or what blocks represent in a system is ambiguous at best. These ambiguities in mapping and meaning hamper formal analysis or a common understanding of SysML v2 models.

G16: Give the user flexibility in how to formulate a model (Walter & Masuch 2011). Being a very general language, SysML v2, similar to UML and SysML v1, gives the modelers complete freedom in formulating models. Once its well-formedness rules are fully developed, this might be restricted.

G17: DSL's support for error prevention and model checking (Kahraman & Bilgen 2015). The current implementation of SysML v2 provides a `SysMLValidator` Xtend class that referential integrity checks, e.g., whether the item usage is typed by an item definition. Besides that, the implementation does not provide any context condition that performs model checking to prevent errors.

G18: Comprehensibility and Learnability (Kahraman & Bilgen 2015). Because SysML v2 has such a wide portfolio of modeling elements and use cases, its documentation has to be similarly comprehensive. The currently available documentation contains a textual definition of each language element and a collection of slides with examples and brief explanations for selected portions. However, the examples miss preciseness and are often too small. Consequently, the basic concept behind an element is understandable, but it is hard to apply to a more sophisticated model.

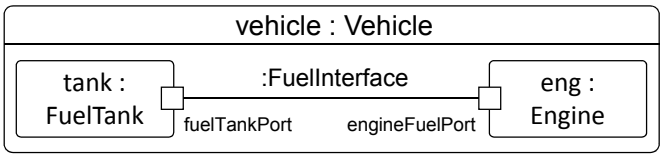

Figure 8 Graphical SysML v2 notation of a part `vehicle` and its features `tank` and `eng` that are connected via the interface `FuelInterface`

G19: The language provides mechanisms for compactness of the representation of the program (Kahraman & Bilgen 2015). The SysML v2 implementation enables to model complex and comprehensive systems. Thus, the models can get complex and comprehensive, too. To mitigate this, SysML v2 provides shorthand notations for, e.g., item flow, binding definitions, state succession, redefinition of parts, and more. This increases development speed and accessibility. Figure 7 shows at the top the definition of a binding between `waterSupply` and `pumpOut` (top, l. 6) and the item flow definition (top, l. 9) separately, and shows the shorthand notation, where both are defined in one statement (bottom, ll. 5-6).

G20: Provide a graphical notation (Jannaber et al. 2017). The SysML v2 provides a graphical notation for most elements of the systems layer of its four-layer language architecture. Elements that do not have a graphical representation are constraints, requirements, analysis cases, verification cases, variation, dependencies, allocation, metadata, element import filtering. Because these modeling elements are usually defined textually only, we do not consider them missing here. Figure 8 shows the graphical notation of a part `vehicle`.

Maintainability

G21: Reuse and compose existing language definitions (Karsai et al. 2009; Völter 2009; Mernik et al. 2005; Kelly & Pohjonen 2009). SysML v2 is developed from scratch. Besides being designed on top of the Kernel Modeling Language (KerML) for basic language foundations (cf. Figure 9), it does not feature any language re-use. Hence, although highly adapting concepts of particular languages, all language constituents must be developed and implemented anew.

G22: Support reusability of language constructs (Karsai et al. 2009; Kelly & Tolvanen 2008; Kelly & Pohjonen 2009). SysML v2 is primarily developed as a big blob within a single grammar[4]. This ultimately prevents reusability in the form of modular sublanguages. Furthermore, there are multiple syntactically related language constructs, which are implemented as individual dedicated productions. This reduces maintainability since changes have to be tracked for many different locations instead of a single one.

G23: Design for language evolution (Völter 2009; Kelly &

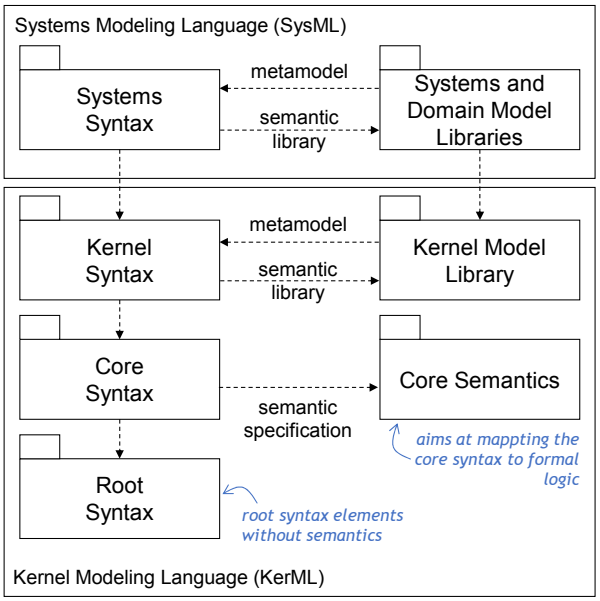

Figure 9 The language architecutre of SysML and KerML[1]

Pohjonen 2009; Livengood 2012; Vierhauser et al. 2015). Similar to the evaluation on **G23**, SysML v2 is not well suited for extensibility and evolution due to its monolithic nature and numerous duplications of production rules for the same syntax constructs.

G24: Reuse type systems (Karsai et al. 2009; Mernik et al. 2005). SysML v2 is not based on any existing language but was developed from scratch, which can be seen in the definition of the language and the realization of the prototypical implementation. Rather, in SysML v2, a type can be created for most model elements. Many of these so-called types lack that which actually makes types, a system of clearly defined rules. Types in SysML v2 instead represent a separation of definition and usage. While the SysML v2 supports the definition of inheritance relationships, other advanced concepts such as generic types or more rigorous type checks and support for type substitution are missing. On the implementation side, the missing reuse of existing type systems is noticeable through the lack of implemented type checks.

G25: Interface concept (Karsai et al. 2009). Blocks and parts in SysML v2 provide ports and thus clearly defined interfaces. Besides that, SysML v2 supports structural inheritance on blocks through which blocks can inherit ports. Therefore, SysML v2 supports a basic interface concept, even if this is not fully fleshed out due to missing semantics, as shown in Figure 6.

G26: Modularize and layer the language (Selic 2009). SysML v2 builds on the novel KerML, as illustrated in Figure 9, featuring foundational concepts, such as identifiers, relations, types, and similar. KerML consists of three languages layered onto another to specify root language elements without semantics, core elements that aim to be mapped to formal logic, and the kernel syntax, which can be extended by libraries. SysML v2 itself is neither modularized nor layered. Hence, a

```
01 // G-29: Three productions with duplicated
02 // code for the same concept of part usages
03 PartUsage returns SysML::PartUsage :
04   UsagePrefix? PartUsageKeyword Usage;
05
06 PartFlowUsage returns SysML::PartUsage :
07   UsagePrefix? 'ref'? PartUsageKeyword Usage;
08
09 PartRefUsage returns SysML::PartUsage :
10   UsagePrefix? ( 'ref' PartUsageKeyword |
11   isComposite ?= PartUsageKeyword ) Usage;
```
Xtext

Figure 10 Excerpt of the SysML v2 grammar impeding variability on language level

decoupled evolution of, e.g., SysML v2 state machines is not possible without further modularization efforts.

G27: Modifiability (Kahraman & Bilgen 2015). The guideline states that modifying the DSL does not degrade its functionality. This corresponds to conservative extension (HKR21 2021). With this, when extending an existing language element, its properties are preserved and thus not restricted. In SysML v2, no mechanisms that ensure conservative extensions are available.

Portability

G28: Support variability on language level (Kelly & Pohjonen 2009). Variant creation for SysML v2 is likely to be cumbersome because it was developed in a single monolithic grammar and has many duplications of productions. For instance, Figure 10 presents three productions for part usages, which mostly substitute each other. This makes it challenging to infer problem-specific adaptations to the language. Although the underlying language workbench Xtext offers some extension mechanisms, these are only rarely used in the current implementation of the language.

G29: Provide for language extensibility (Selic 2009). Providing for language extensibility means providing mechanisms (such as interfaces (BPR+20 2020), extension points (HRW18 2018), hook points, or similar) for the systematically planned extension of the language and its elements. For SysML v2, such mechanisms are not planned. Instead, the language and its Xtext reference implementation can be reused opportunistically, e.g., by creating new languages extending SysML v2 and adding/removing elements as desired.

```
01 // G-31: Action definition with embedded
02 // statements of a foreign language
03 action def CreateFile(in size : Integer) {
04   language "Java"
05   /* File f = new File(size);
06    * f.createNewFile();
07    */
08 }
```

Figure 11 SysML v2 uses "opaque actions" to integrate models of other languages as comments.

G30: Allow for incorporating foreign language fragments in models (Selic 2009). SysML v2 supports the notion of "opaque actions", which are action definitions that feature a body of elements of another language in the form of multi-line comments. Content in these comments is treated as plain text and is unaccessible for, e.g., well-formedness checking. Figure 11 illustrates this mechanism on a SysML v2 action definition featuring Java code embedded in a comment. Note that this excerpt is not well-formed as the `File()` constructor (l. 5) does not accept integer arguments. This issue, however, cannot be detected automatically at design time as SysML v2 cannot check the "embedded" Java parts.

4. Analysis Results

This section summarizes our findings on the language engineering of SysML v2. To this end, Table 1 summarizes the compliance of SysML v2 with the presented SLE guidelines.

Overall, we determined that SysML v2 fulfills eleven of the applicable guidelines completely, seven other guidelines at least partially, and 12 guidelines not at all. While most of the guidelines regarding functional suitability and usability are fulfilled or partially fulfilled, maintainability and portability might require further considerations.

If SysML v2 achieves similar popularity and use than SysML v1 or UML, researchers and practitioners will employ the language for largely different purposes and use cases. We expect that this will lead to the proliferation of variants as it did for the other languages, which produced variants of UML, such as DiSpa (Bergert et al. 2007), MechatronicUML (Burmester et al. 2004), UMM (Hofreiter et al. 2006), and UML4IoT (Thramboulidis & Christoulakis 2016) or variants of SysML, such as SysML4Mechatronics (Li et al. 2019) and SysML4Modelica (Reichwein et al. 2012). Thus, the portability and maintainability of the SysML v2 language are essential properties to adjust and evolve with changing requirements from different domains. Currently, extending, tailoring, or restricting SysML v2 to specific domains demands opportunistic changes to the language and its tooling by language engineering experts. Realizing SysML v2 as it will be standardized on the foundations of established language workbenches (Erdweg et al. 2013; HRW18 2018) could help to keep SysML v2 relevant in the presence of the changing requirements on the language.

Other language guidelines are not fulfilled by SysML v2 due to lacking semantics (as in "meaning" (Harel & Rumpe 2004); neither behavior nor well-formedness). For instance, neither starting with a semantics model (**G2**), nor deriving concepts from the expected output (**G13**), or having a clear language to target mapping (**G15**) are currently supported by SysML v2. We are aware that there is ongoing work on the semantics and wish for the work to succeed, as a SysML v2 without semantics would again be a (textual) sketching tool only.

5. Proposed Improvements

Overall, we identified different aspects in the current release of the SysML v2 implementation that can be improved. While some guidelines yield a trade-off (e.g., avoiding redundancy

Table 1 Compliance (C.) of SysML v2 with applicable guidelines (● = fulfilled, ◑ = partially fulfilled, ○ = not fulfilled)

No.	Guideline	C.
Functional Suitability		
G1	Balance generality and specificity (Karsai et al. 2009; Völter 2009; Wile 2004)	●
G2	Always start with a semantics model (Selic 2009)	○
G3	Defined scope and purpose (Jannaber et al. 2017)	●
G4	The modeling language is specified by a language metamodel (Jannaber et al. 2017)	◑
Compatibility		
G5	Provide integrability (Kolovos et al. 2006; Kelly & Tolvanen 2008)	●
Usability		
G6	Adopt existing domain notations (Karsai et al. 2009; Kelly & Tolvanen 2008; Völter 2009; Wile 2004; Mernik et al. 2005; Frank 2013)	●
G7	Avoid redundancy (Karsai et al. 2009; Kolovos et al. 2006; Salehi et al. 2016)	◑
G8	Viewpoint orientation (Kahlaoui et al. 2008; Kelly & Tolvanen 2008; Völter 2009)	●
G9	Make elements distinguishable (Karsai et al. 2009; Kelly & Tolvanen 2008; Kelly & Pohjonen 2009)	●
G10	Consistent style everywhere (Karsai et al. 2009; Kelly & Tolvanen 2008)	●
G11	Define language rules (Kelly & Tolvanen 2008)	◑
G12	First things first - the language (Kelly & Tolvanen 2008)	●
G13	Derive concept from expected output (Kelly & Tolvanen 2008)	○
G14	Multiple level of abstractions (Frank 2013)	◑
G15	Clear language to target mapping (Frank 2013)	○
G16	Give the user flexibility in how to formulate a model (Walter & Masuch 2011)	●
G17	DSL's support for error prevention and model checking (Kahraman & Bilgen 2015)	○
G18	Comprehensibility and Learnability (Kahraman & Bilgen 2015)	◑
G19	The language provides mechanisms for compactness of the representation of the program (Kahraman & Bilgen 2015)	●
G20	Provide a graphical notation (Jannaber et al. 2017)	●
Maintainability		
G21	Reuse and compose existing language definitions (Karsai et al. 2009; Völter 2009; Mernik et al. 2005; Kelly & Pohjonen 2009)	○
G22	Support reusability of language constructs (Karsai et al. 2009; Kelly & Tolvanen 2008; Kelly & Pohjonen 2009)	○
G23	Design for language evolution (Völter 2009; Kelly & Pohjonen 2009; Livengood 2012; Vierhauser et al. 2015)	○
G24	Reuse type system (Karsai et al. 2009; Mernik et al. 2005)	○
G25	Interface concept (Karsai et al. 2009)	◑
G26	Modularize and layer the language (Selic 2009)	○
G27	Modifiability (Kahraman & Bilgen 2015)	○
Portability		
G28	Support variability on language level (Kelly & Pohjonen 2009)	○
G29	Provide for language extensibility (Selic 2009)	○
G30	Allow for incorporating foreign language fragments in models (Selic 2009)	◑

increases the learnability but mitigates the modelers' productivity), there are others from which we can directly derive sound suggestions for improvement.

One of the most striking aspects is the lack of semantics. While **G2** even suggests starting with the semantics model, the SysML v2 specification should at least aim at providing it in the long run. Without a formal description of the provided elements' meaning, modelers cannot create sound models. In fact, models would be created with a subjective interpretation meaning, resulting in inconsistencies among different users. While this already poses a problem for ordinary modeling languages, it would be fatal in the case of SysML v2 since the intended interdisciplinarity would be subverted by construction. Additionally, semantics is required for more sophisticated applications such as verification or code generation.

Well-formedness rules (**G11**) are a crucial part of language definitions. They support modelers by automated validation of constraints that context-free grammars cannot cover (HKR21 2021). The implementation of SysML v2 supports only a few well-formedness rules. Especially sophisticated type checks and constraints about which elements may (or should) be used in a particular context still reveal significant gaps. As the SysML v2 has not been standardized yet, we expect that some missing rules will be provided in future releases. Generally, however, well-formedness of system models highly depends on their purpose (communication, documentation, code generation, ...) as well as on their domain, e.g., due to certification challenges, avionics usually has stricter well-formedness requirements than Industry 4.0. Thus, we can expect modelers needing to tailor the well-formedness rules of "their" SysML v2. As SysML v2 does not provide dedicated extension mechanisms for this, such application-specific and domain-specific rules must be developed using the mechanisms of Xtext (Bettini 2016).

Furthermore, it is apparent that SysML v2 is only conceptually based on existing languages but not concerning its implementation (**G21**). There are numerous language components, e.g., for expressions (Büttner & Gogolla 2011), types (BEH+20 2020) (**G24**), and statements (OMG 2017), that would be excellent foundations for SysML v2 instead of developing these language parts from scratch. Additionally, as SysML v2 partially reinterprets existing languages, there is a high potential for reusing these as well (e.g., state machines (*Yakindu Statechart Tools* n.d.)). Reusing existing modeling languages not only reduces the development time by building upon tried-and-tested language modules but also can increase comprehension and acceptance as modelers may recognize reused concepts.

Also, lack of modularization (**G26**) might pose challenges in the future of SysML v2: Its current monolithic realization hinders variability (**G28**) and extensibility (**G29**), which might impede the overall portability of the language. Thus, we propose a more modular approach, e.g., by following the language boundaries introduced by the four pillars of SysML (Hause 2006). Separating distinct language components fosters modifiability and independent evolution of particular aspects. Furthermore, the overall learnability could be increased by exposing novice modelers with selected/required language constituents only (e.g., only structural modeling elements). To support this notion, many frameworks, such as MetaEdit+ (Kelly et al. 1996), MontiCore (HKR21 2021), MPS (Voelter & Pech 2012), Spoofax (Kats & Visser 2010), and Xtext (Bettini 2016), support various language composition techniques. Composability has been studied extensively (Erdweg et al. 2013), thus constituting a solid foundation for modularization.

For the integration of foreign language fragments into SysML v2 models (**G30**), language engineering has produced various means, such as weaving (Degueule et al. 2015) or embedding (HRW18 2018) parts of a language into a host language. These mechanisms combine the abstract syntax of both languages such that well-formedness checking of the integrated models becomes possible and, hence, errors can be identified early and reliably.

6. Discussion

The presented findings are based on an analysis of state-of-the-art language design guidelines available in the literature. Based on multiple guidelines, we selected those relevant for SysML v2 and performed an evaluation of the language. The obtained insights should help to improve the currently developing language as well as potential variants.

Following a typical study design, our contribution is consequently subject to threats to validity. According to (Wohlin et al. 2012), we classify and analyze these threats as construct, internal, external, and reliability validity. Construct validity is based on the study's design. External validity represents the generalizability of a study, while internal threats influence the inferred conclusions, i.e., its specificity. Reliability refers to the trustworthiness and reproducibility of a study's results.

A possible threat regarding construct validity arises from the approach for retrieving the language design guidelines. To obtain in-depth information about current best practices, we started with another recent mapping study (Czech et al. 2018) and extracted the set of guidelines by backward and forward snowballing. The results were filtered and clustered with respect to ISO 25010 (ISO/IEC 2010). While this procedure yields a profound set of guidelines, some unrelated studies may cover further best practices.

While the analysis results of our work are naturally specific to SysML v2, the elaborated guidelines are generalizable to a certain degree as they result from SLE requirements. However, we filtered the guidelines with respect to language design, which affects the external validity. Thus, guidelines that refer to the overall tooling or development process, such as providing an appropriate syntax highlighting or including stakeholder groups during development, were not considered. Furthermore, some guidelines are, per se, not falsifiable. For instance, while most will agree that modeling languages should be designed for longevity, usability, and productivity, these aspects are hardly measurable. Thus, we excluded these practices as well.

Natural threats to internal validity are the conclusions drawn for the particular guidelines. To reduce this effect, we thoroughly discussed all results among the authors. Similarly, this also impacts the study's reliability, as other readers might infer slightly different conclusions. Finally, it is essential to mention

that SysML v2 is still under development. Thus, some of our results could be resolved in future releases.

7. Related Work

While there has been comprehensive literature on language design guidelines and best practice, their application to specific languages rarely is documented. For languages going through consortium-based standardization, post-hoc analysis of their design often is futile as one cannot expect to take the results of such analysis into account. Another reason for this lack of research might be that most languages and language variants are used by (smaller) communities only, which have not been considered by language engineers yet. Consequently, we neither found applications of DSL guidelines to widely used languages, such as AutomationML, UML, Simulink, or SysML v1, nor to less popular modeling languages or DSLs. However, several publications investigate the use of modeling languages, but we are aware of none exploring their adherence to language guidelines. A notable exception is a discussion of the foundations of UML (Cook 2012), which discusses challenges in its semantic foundations. With SysML v2 being conceived independently of UML, discussions of UML, however, are of limited relevance for its design.

8. Conclusion

We have presented an analysis of the current state of the language design of SysML v2. For this, we first gathered language design guidelines from several publications. Then, we clustered all considered guidelines into categories compliant with ISO-25010, which categorizes software quality in various characteristics. Applied to the SysML v2, we analyzed the compliance of the guidelines and differentiated between fulfilled, not fulfilled, or partly fulfilled in our results. We found that SysML v2, in its current 2022-01 release, fulfills many guidelines regarding functional suitability, compatibility, and usability. However, maintainability and portability both need further improvement and adjustment. At its current state, it is time-consuming to tailor, extend or restrict SysML v2 to fulfill the requirements of specific domains. Therefore, we suggest modularizing the language for future releases. Also, the lack of a semantic mapping hinders modelers from understanding the meaning of modeling elements.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612

References

Andrianarison, E., & Piques, J.-D. (2010). SysML for embedded automotive Systems: a practical approach. In *ERTS2 2010, Embedded Real Time Software & Systems.*

Bergert, M., Diedrich, C., Kiefer, J., & Bar, T. (2007). Automated PLC software generation based on standardized digital process information. In *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)* (pp. 352–359).

Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd.

Blanchard, B. S., Fabrycky, W. J., & Fabrycky, W. J. (1990). *Systems Engineering and Analysis* (Vol. 4). Prentice hall Englewood Cliffs, NJ.

Bone, M., & Cloutier, R. (2010). The Current State of Model Based Systems Engineering: Results from the OMG™ SysML Request for Fnformation 2009. In *Proceedings of the 8th Conference on Systems Engineering Research.*

Broy, M., & Stølen, K. (2001). *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer Verlag Heidelberg,.

Bucchiarone, A., Ciccozzi, F., Lambers, L., Pierantonio, A., Tichy, M., Tisi, M., … Zaytsev, V. (2021). What is the future of modeling? *IEEE software*, *38*(2), 119–127.

Buede, D. M., & Miller, W. D. (2016). *The Engineering Design of Systems: Models and Methods*. John Wiley & Sons.

Burmester, S., Giese, H., & Tichy, M. (2004). Model-driven development of reconfigurable mechatronic systems with mechatronic UML. In *Model Driven Architecture* (pp. 47–61). Springer.

Butting, A., Eikermann, R., Hölldobler, K., Jansen, N., Rumpe, B., & Wortmann, A. (2020, October). A library of literals, expressions, types, and statements for compositional language design. *Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology*, *19*, 3:1-16. (Special Issue dedicated to Martin Gogolla on his 65th Birthday)

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., & Wortmann, A. (2018, January). Controlled and extensible variability of concrete and abstract syntax with independent language features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)* (pp. 75–82). ACM.

Butting, A., Kautz, O., Rumpe, B., & Wortmann, A. (2019, March). Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software*, *149*, 437–461. doi: https://doi.org/10.1016/j.jss.2018.12.016

Butting, A., Pfeiffer, J., Rumpe, B., & Wortmann, A. (2020, October). A compositional framework for systematic modeling language reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (p. 35–46). ACM.

Butting, A., Rumpe, B., & Wortmann, A. (2016, October). Embedding component behavior DSLs into the MontiArcAutomaton ADL. In *Globalization of Modeling Languages Workshop (GEMOC'16)* (Vol. 1731).

Büttner, F., & Gogolla, M. (2011). Modular Embedding of the Object Constraint Language into a Programming Language. In *Brazilian Symposium on Formal Methods* (pp. 124–139).

Campagne, F. (2014). *The MPS language workbench: volume I* (Vol. 1). Fabien Campagne.

Cawasji, K. A., & Baras, J. S. (2018). SysML Executable Model of an Energy-Efficient House and Trade-Off Analysis.

In *2018 IEEE International Systems Engineering Symposium (ISSE)* (pp. 1–8).

Cengarle, M. V., Grönniger, H., & Rumpe, B. (2009). Variability within modeling language definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)* (pp. 670–684). Springer.

Clark, T., Brand, M. v. d., Combemale, B., & Rumpe, B. (2015). Conceptual model of the globalization for domain-specific languages. In *Globalizing Domain-Specific Languages* (pp. 7–20). Springer.

Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., Steel, J., & Vojtisek, D. (2016). *Engineering modeling languages: Turning domain knowledge into tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series.

Combemale, B., Kienzle, J., Mussbacher, G., Barais, O., Bousse, E., Cazzola, W., … Wortmann, A. (2018). Concern-oriented language development (COLD): Fostering reuse in language engineering. *Computer Languages, Systems & Structures*, *54*, 139 - 155.

Cook, S. (2012). Looking back at UML. *Software & Systems Modeling*, *11*(4), 471–480.

Czech, G., Moser, M., & Pichler, J. (2018). Best Practices for Domain-Specific Modeling. A Systematic Mapping Study. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (p. 137-145). doi: 10 .1109/SEAA.2018.00031

Degueule, T., Combemale, B., Blouin, A., Barais, O., & Jézéquel, J.-M. (2015). Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*. Pittsburgh, United States.

Degueule, T., Mayerhofer, T., & Wortmann, A. (2017, September). Engineering a ROVER Language in GEMOC Studio & MontiCore: A Comparison of Language Reuse Support. In *Proceedings of MODELS 2017. Workshop EXE.*

de Kinderen, S. (2017). Using grounded theory for domain specific modelling language design. In *IFIP Working Conference on The Practice of Enterprise Modeling* (pp. 34–48).

Dévai, G., Kovács, G. F., & An, Á. (2014). Textual, Executable, Translatable UML. In *OCL@ MoDELS* (pp. 3–12).

Erdweg, S., Van Der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., … Loh, A. (2013). The state of the art in language workbenches. In *International Conference on Software Language Engineering* (pp. 197–217).

Forsythe, C. (2013). *Instant FreeMarker Starter*. Packt Publishing Ltd.

Frank, U. (2013). Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines. In *Domain engineering* (pp. 133–157). Springer.

Grönniger, H., & Rumpe, B. (2011). Modeling language variability. In *Workshop on modeling, development and verification of adaptive systems* (pp. 17–32). Springer.

Group, O. M. (2017). *Systems Modeling Language (SysML®) v2 Request For Proposal (RFP).* Retrieved 2021-05-05, from https://www.omg.org/cgi-bin/doc.cgi?ad/2017-12-2

Hamilton, M. H., & Hackler, W. R. (2007). 8.3.2 A Formal Universal Systems Semantics for SysML. In *INCOSE International Symposium* (Vol. 17, pp. 1333–1357).

Harel, D., & Rumpe, B. (2004, October). Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, *37*(10), 64–72.

Hause, M. (2006). The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference* (Vol. 9, pp. 1–12).

Hayden, J. L., & Jeffries, A. (2012). On Using SysML, DoDAF 2.0 and UPDM to Model the Architecture for the NOAA's Joint Polar Satellite System (JPSS) Ground System (GS). In *12th International Conference on Space Operations.*

Heim, R., Mir Seyed Nazari, P., Rumpe, B., & Wortmann, A. (2016, July). Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)* (pp. 67–82). Springer.

Herzig, S. J., Karban, R., Trancho, G., Dekens, F., Jankevicius, N., & Troy, M. (2017). Analyzing the Operational Behavior of the Alignment and Phasing System of the Thirty Meter Telescope using SysML. In *Proceedings of the Conference on Adaptive Optics for Extremely Large Telescopes 5.*

Hoare, C. A. R. (1978). Communicating Sequential Processes. *Communications of the ACM*, *21*(8), 666–677.

Hofreiter, B., Huemer, C., Liegl, P., Schuster, R., & Zapletal, M. (2006). UN/CEFACT'S modeling methodology (UMM): a UML profile for B2B e-commerce. In *International Conference on Conceptual Modeling* (pp. 19–31).

Hölldobler, K., Kautz, O., & Rumpe, B. (2021). *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag.

Hölldobler, K., Roth, A., Rumpe, B., & Wortmann, A. (2017, October). Advances in Modeling Language Engineering. In *International Conference on Model and Data Engineering* (pp. 3–17). Springer.

Hölldobler, K., Rumpe, B., & Wortmann, A. (2018). Software language engineering in the large: Towards composing and deriving languages. *Computer Languages, Systems & Structures*, *54*, 386–405.

ISO/IEC. (2010). *ISO/IEC 25010 system and software quality models* (Tech. Rep.). Madrid, Spain: International Standardization Organization (ISO).

Jannaber, S., Riehle, D. M., Delfmann, P., Thomas, O., & Becker, J. (2017). Designing a framework for the development of domain-specific process modelling languages. In *International Conference on Design Science Research in Information System and Technology* (pp. 39–54).

Jolak, R., Savary-Leblanc, M., Dalibor, M., Wortmann, A., Hebig, R., Vincur, J., … Chaudron, M. R. V. (2020, November). Software engineering whispers : The effect of textual vs. graphical software design descriptions on software design communication. *Empirical software engineering*, *25*(6), 4427–4471.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., & Valduriez, P. (2006). ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*

(pp. 719–720).

Kahlaoui, A., Abran, A., & Lefebvre, E. (2008). DSML Success Factors and their Assessment Criteria. *Metrics News*, *13*(1), 43–51.

Kahraman, G., & Bilgen, S. (2015). A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, *14*(4), 1505–1526.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., & Völkel, S. (2009, October). Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)* (pp. 7–13). Helsinki School of Economics.

Kats, L. C., & Visser, E. (2010). The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (pp. 444–463).

Kautz, O., Rumpe, B., & Wortmann, A. (2020, April). Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, *27*, 119-151.

Kelly, S., Lyytinen, K., & Rossi, M. (1996). Metaedit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *International Conference on Advanced Information Systems Engineering* (pp. 1–21).

Kelly, S., & Pohjonen, R. (2009). Worst Practices for Domain-Specific Modeling. *IEEE software*, *26*(4), 22–29.

Kelly, S., & Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons.

Kolovos, D. S., Paige, R. F., Kelly, T., & Polack, F. A. (2006). Requirements for Domain-Specific Languages. In *Proc. of ECOOP Workshop on Domain-Specific Program Development (DSPD)* (Vol. 2006).

Kossiakoff, A., Sweet, W. N., Seymour, S. J., & Biemer, S. M. (2011). *Systems Engineering Principles and Practice* (Vol. 83). John Wiley & Sons.

Li, H., Zou, M., Weidmann, D., Cheaib, S. A., Mörtl, M., & Vogel-Heuser, B. (2019). Model-based Systems Engineering Process for Supporting Variant Selection in the Early Product Development Phase. In *2019 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)* (pp. 637–643).

Lima, L., Didier, A., & Cornélio, M. (2013). A formal semantics for SysML activity diagrams. In *Brazilian Symposium on Formal Methods* (pp. 179–194).

Livengood, S. (2012). Experiences in Domain-Specific Modeling for Interface Specification and Development. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards* (pp. 1–2).

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and How to Develop Domain-Specific Languages. *ACM computing surveys (CSUR)*, *37*(4), 316–344.

OMG. (2011, August). *OMG unified modeling language (OMG UML), superstructure, version 2.4.1*.

OMG. (2017, June). *Action Language for Foundational UML (ALF)*.

OMG. (2021). *Semantics of a foundational subset for executable UML models (fUML) - Version 1.5* (Tech. Rep.). Milford, United States: Object Management Group (OMG).

Petrausch, V., Seifermann, S., & Müller, K. (2016). Guidelines for accessible textual UML modeling notations. In *International Conference on Computers Helping People with Special Needs* (pp. 67–74).

Reichwein, A., Paredis, C. J., Canedo, A., Witschel, P., Stelzig, P. E., Votintseva, A., & Wasgint, R. (2012). Maintaining consistency between system architecture and dynamic system models with SysML4Modelica. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling* (pp. 43–48).

Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International.

Salehi, P., Hamou-Lhadj, A., Toeroe, M., & Khendek, F. (2016). A UML-based Domain Specific Modeling Language for Service Availability Management: Design and Experience. *Computer Standards & Interfaces*, *44*, 63–83.

Sangiorgi, D., & Walker, D. (2003). *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press.

Selic, B. (2009). The theory and practice of modeling language design for model-based software engineering—a personal perspective. In *International Summer School on Generative and Transformational Techniques in Software Engineering* (pp. 290–321).

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

Thramboulidis, K., & Christoulakis, F. (2016). UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry*, *82*, 259–272.

Vierhauser, M., Rabiser, R., Grünbacher, P., & Egyed, A. (2015). Developing a DSL-Based Approach for Event-Based Monitoring of Systems of Systems: Experiences and Lessons Learned (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 715–725).

Voelter, M., & Pech, V. (2012). Language modularity with the MPS language workbench. In *2012 34th International Conference on Software Engineering (ICSE)* (pp. 1449–1450).

Völter, M. (2009). Best Practices for DSLs and Model-Driven Development. *Journal of Object Technology*, *8*(6), 79–102.

Walter, R., & Masuch, M. (2011). How to integrate domain-specific languages into the game development process. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology* (pp. 1–8).

Wile, D. (2004). Lessons Learned from Real DSL Experiments. *Science of Computer Programming*, *51*(3), 265-290.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Science & Business Media.

Wymore, A. W. (2018). *Model-Based Systems Engineering* (Vol. 3). CRC press.

Yakindu Statechart Tools. (n.d.). https://www.itemis.com/en/yakindu/state-machine/. (Accessed: 2021-07-07)

Zaytsev, V. (2014). Grammar Maturity Model. In *ME@MODELS* (pp. 42–51).

About the authors

Nico Jansen is a research assistant at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, software architectures, and model-based software and systems engineering. You can contact the author at jansen@se-rwth.de or visit https://www.se-rwth.de/staff/jansen/.

Jérôme Pfeiffer is a research assistant at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart. His research interests include software language engineering techniques and applied model-driven engineering with a focus on digital twins and Industry 4.0. You can contact the author at jerome.pfeiffer@isw.uni-stuttgart.de or visit https://www.isw.uni-stuttgart.de/en/institute/team/Pfeiffer-00005/.

Bernhard Rumpe is a professor heading the Software Engineering department at the RWTH Aachen University, Germany. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model-engineering based on UML/SysML-like notations and domain-specific languages. You can contact the author at rumpe@se-rwth.de or visit https://www.se-rwth.de/staff/rumpe/.

David Schmalzing is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software architectures, model-driven software development, and model-based systems engineering. You can contact the author at schmalzing@se-rwth.de or visit https://www.se-rwth.de/staff/schmalzing/.

Andreas Wortmann is a professor at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart where he conducts research on model-driven engineering, software language engineering, and systems engineering with a focus on Industry 4.0 and digital twins. You can contact the author at andreas.wortmann@isw.uni-stuttgart.de or visit www.wortmann.ac.