Achievements, Failures, and the Future of Model-Based Software Engineering

Oliver Kautz, Alexander Roth, and Bernhard Rumpe

The borders of my language are the borders of my world.
L. Wittgenstein

1 Introduction

Using models is one of the primary techniques to understand and engineer the world. Modeling is by far not an invention of software engineering. All engineering disciplines use models to describe a system under development before actually building it. A model is used to get a shared understanding of the system and also to analyze whether the system will have the desired properties after building it. The term "model" dates back to the twelfth century, where a model meant to be a 1:10 version of a cathedral or dome, allowing a customer to "walk through" the building, to understand whether size, light effects, impressiveness, or other properties will be achieved, but also to understand whether the building will be statically save.

Models are not only used as a prescription of a system to be built. Models are also used in science to understand the existing world in a descriptive form. Already the ancient Greeks and Egyptians have built models of their worlds, including mathematical laws and a calendar system to predict the monsoon—it is barely imaginable that a pyramid could be built without extensive prior modeling. Archaeologists believe that cave paintings were used to teach younger tribal members hunting herds of wild animals. These paintings are models of hunting scenes. In social communities we learn role models for our behavior. We also get executable models in forms of recipes for preparing meals or construction manuals for assembling furniture.

Models appear in a plethora of forms. They can describe structural or behavioral aspects, interactions, and geometry, or they can be used as recipes for processes. Models can be quite informal and self-explanatory (e.g., cooking recipe, furniture

O. Kautz (✉) · A. Roth · B. Rumpe
Software Engineering, RWTH Aachen University, Aachen, Germany
e-mail: kautz@se-rwth.de; roth@se-rwth.de; rumpe@se-rwth.de

V. Gruhn, R. Striemer (eds.), *The Essence of Software Engineering*,
https://doi.org/10.1007/978-3-319-73897-0_13

construction manual) or have a rather formal appearance and a well-defined mathematical theory (e.g., differential equations for physics simulations).

With all this different forms of models in society, science, and engineering, it is not surprising that computer science developed its own idea of helpful models to design and understand software systems.

However, software is different from any other physical system that engineers would want to build. First of all, software per se has no physical manifestation. This leads to a number of characteristics for software itself as well as for software production and development processes. Obviously, production completely vanishes, when software can be downloaded and built on a push-button basis (or even automatically). There is no need for a physical manifestation of a factory for assembling individual software products during a rather expensive process. Therefore, software development is always a process of invention and stabilization until the final version is developed.

Because of its immaterial nature and the knowledge about its context, software itself is (or at least it embodies) a model of the elements that it deals with. A "Person" object is actually an abstraction of a real Person, containing only the relevant information about the person. But software is complex and thus needs engineering techniques. The two main classical techniques are "divide and conquer" in various forms: The software system is divided in subsystems and finally components. The development process is divided in phases, focusing on different artifacts, and iterations, allowing to start small and to improve the software in manageable steps. Early phases, such as understanding the problem (also called requirements elicitation), structuring the problem (also called requirements specification and high-level architecture definition), as well as precisely specifying the desired solution in smaller chunks (e.g., use case definition), do not result in programming artifacts but require various forms of models.

The Unified Modeling Language (UML) [39, 43, 44] was developed two decades ago to provide a standardized framework for this purpose. It includes and standardizes preexisting modeling approaches and languages and adds new ones. The UML has 13 sublanguages in total that are applicable for a variety of purposes. However, it is difficult to say whether the UML, as a general-purpose modeling language, is the appropriate language for all use cases or where it should be improved or where language concepts should be removed or added.

The UML, its many predecessors, and its foundations in modeling techniques based on formal methods, such as finite automata [24], entity relationship models [9], and others, demonstrate that computer science has not only invented its own forms of models but also made its forms of models explicit through defining modeling languages. Only if a modeling language is defined explicitly and precisely enough, which includes syntax, semantics, and the clarification about the pragmatic forms of use [22], then the language is usable for communication, shared understanding between humans, and also amenable for intensive tooling, such as checking semantic differences [30, 31] or consistency analysis [32].

In general, MBSE includes all development processes that use explicit models in artifacts, both for internal communication as well as for communication with a computer. The latter is intended for detailed analysis, simulation, productive or test code generation, interpretation, and configuration.

Subsequently, this chapter first clarifies in more detail what MBSE is and what the most popular modeling languages are in Sect. 2. Afterwards, we discuss its current failures and achievements in Sects. 3 and 4 to conclude with an outlook of the future of MBSE in Sect. 5.

2 Model-Based Software Engineering

MBSE is a software development process that aims to tackle increasing software development complexity by using abstraction and automation [5]. Abstraction is achieved by employing suitable models of (parts of) a software system. Automation systematically transforms these models into executable source code. The term model is considered as a high-level abstraction in textual or graphical notation. Even though its meaning is not clearly defined (cf. [45]), we understand a model that is used in MBSE as follows:

Definition 1 (Model [12]) A model is an abstraction of an aspect of reality (as-is or to-be) that is built for a given purpose.

In general, models can be distinguished into prescriptive and descriptive models depending on their purpose and use [17]. The primer describe an original that is created from the model. The latter describe an original to better understand it. Disregarding this classification, each model has to be written down in order to be useful for MBSE. This already implies that models in MBSE have finite representations, either in textual or graphical notation. Specifying models can either be done using a general-purpose modeling language (GPML) or a domain-specific language (DSL).

Definition 2 (Modeling Language [12]) A modeling language defines a set of models that can be used for modeling purposes. Its definition consists of (a) the syntax, (b) the semantics, and (c) its pragmatics.

While modeling languages are usually not tailored to a particular domain but rather address general-purpose concepts (e.g., the UML [39]), a DSL uses wording and concepts from the domain of interest. Hence, we understand a DSL as follows:

Definition 3 (Domain-Specific Language [11]) A DSL is a language that is specifically dedicated to a domain of interest, where a language is understood as a means for communication between stakeholders using a restricted amount of sentences.

A DSL targets at bridging the gap between problem and solution space [11] and is, generally, more restrictive than a GPML. DSLs usually drop Turing completeness and often allow fully automated formal verification of the (domain-specific) properties of interest. This is hardly feasible using Turing-complete general-purpose programming languages (GPLs).

2.1 *Unified Modeling Language and Systems Modeling Language*

A standardized language family for software development is the Unified Modeling Language, which is standardized by the Object Management Group [39]. The UML offers a wide portfolio of mainly graphical (but also textual) languages to model software systems [51]. An overview of all diagram types is shown in Fig. 1.

While the primary focus of the UML targets at software systems' design, another popular language (the Systems Modeling Language (SysML)) aims to provide a similar set of modeling languages for the design of systems [48]. It is based on the UML 2 and, hence, uses some diagrams with particular extensions for system design. In addition, the SysML provides the requirement diagram type, which is not part of the UML, to address requirements engineering. An overview of all supported diagram types is shown in Fig. 2.

2.2 *Constructive Model Use: Interpreters and Code Generators*

One major goal of MBSE is to model a software system or parts of the system by abstract models describing a subject of a particular problem domain. Models are

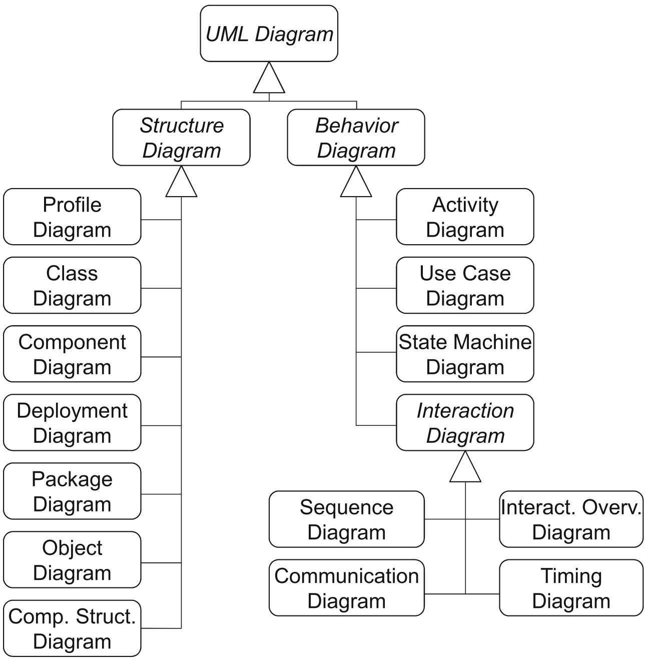

Fig. 1 Overview of UML diagram types

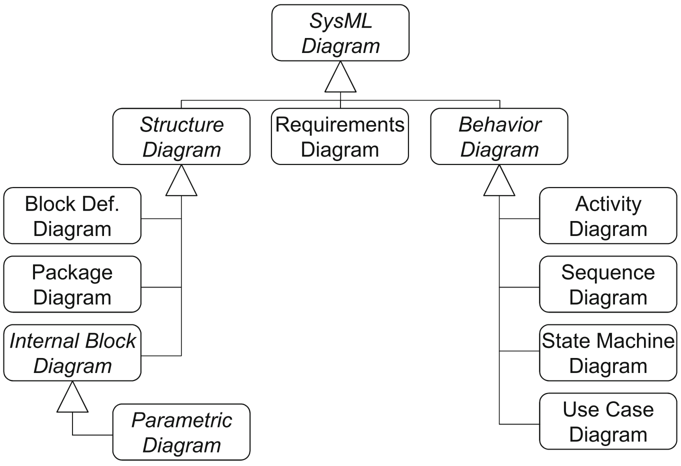

Fig. 2 Overview of SysML diagram types

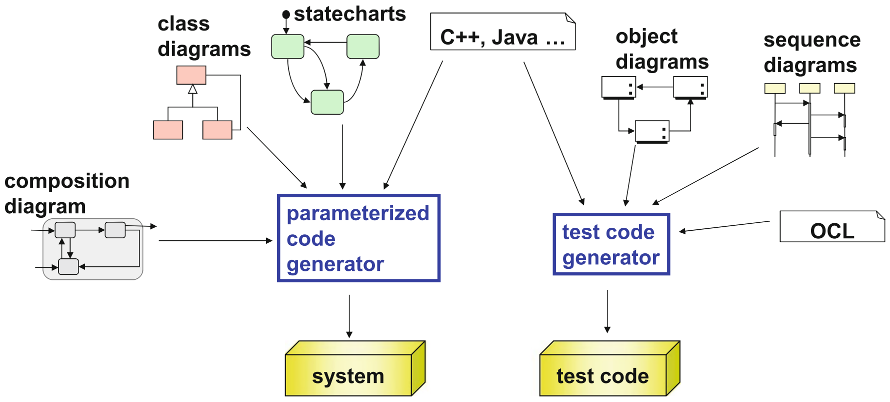

Fig. 3 Overview of different scenarios for using models in MBSE to generate system or test code

then used constructively for different aspects of a software system. This is achieved by developing and employing code generators as well as model interpreters to reflect the models' meanings in a system, which is potentially still under development.

A code generator takes models as input and produces (parts of) a software system [42]. Assuming code generator's correctness, the intended benefit is to reduce the costs for developing the system by hand. As models typically omit certain details, the generated code typically has to be complemented with handwritten code that the model abstracts from. This can either be done on the generated source code level (e.g., [21]) or on the input model level (e.g., UML/P [45]).

Figure 3 gives an overview of how UML models can be employed in MBSE. For example, UML class diagrams can be used to generate system code, which is source code used in production. Another example are UML object diagrams that can be used to define test cases and generate test code (cf. [44]). The main part of both examples is formed by a code generator to systematically transform a model into executable source code.

As an alternative to code generators, interpreters can be used to execute models. Interpreters are kinds of software systems that operate on models in context of a running system, that is, they interpret models based on a system's state. As the system state may change (possibly also during the interpretation of a model), the result from interpreting the same model may differ between successive interpretations. In contrast to code generation, model interpretation relies on a virtual machine that is aware of the semantics of the model. When executing a model by interpretation, a model interpreter evaluates the model and executes the corresponding virtual machine commands. Such approaches are known from scripting languages such as Python or Ruby, even though they do not conceptually differ from the underlying programming paradigm and are programming languages, too.

Some major benefits of code generation over interpretation are that (a) generated source code is easier to understand than interpreter logic, (b) generated source code can be optimized for different target platforms, and (c) code generators are easier to start with and to maintain because of existing code generator frameworks. In contrast to code generation, the major benefits of model interpretation are as follows: (a) changes in a model do not require code generation and, hence, allow higher agility in development, (b) changes to a model can be made at runtime and are directly reflected by the interpreter [10], and (c) it promises to be more flexible than code generation. Stated differently, code generation is typically performed at design time, before compile time, whereas model interpretation is typically done at runtime.

2.3 *Benefits and Drawbacks of Using Models Constructively*

Models explicitly describe a particular subject from the domain of interest and implicitly describe (parts of) a system for solving the problem the model describes. This implicit description is made explicit by code generators or interpreters.

Figure 4 illustrates the relations between models, the domain-specific problems they describe, and the solutions for solving the problems. Each model explicitly models (parts of) a domain-specific problem. When using the model constructively, it also implicitly relates the parts of the solution to the parts of the problem it describes. A code generator explicates this relation by defining the translation rules from the model to a representation in the solution space (typically GPL code) for solving the parts of the problem described by the model. Similarly, an interpreter explicates the description by defining the necessary steps for executing the model at runtime to solve the parts of the problem the model describes.

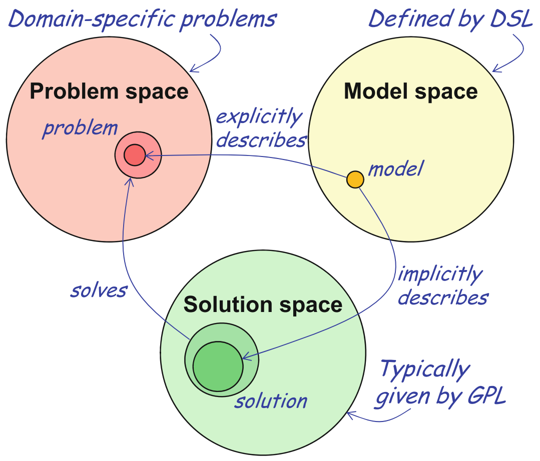

Fig. 4 Relation between domain-specific problems, models addressing the domain, and solutions to the problems

As models are intended to be more abstract than the subject they describe (problem space), models are also considerably easier to understand than the parts of the system they describe (solution space). They simultaneously serve as system documentation (solution space description) and primary development artifacts (problem space description). This avoids redundancies between system documentations and inconsistencies between documentations and implementations. Constructive use of models further significantly reduces the cost of system evolution: Adjusting a model can be carried out more effectively than adjusting the system it describes, which should be much more complex in comparison to the model. A model's syntax is typically expressed in the terminology of its application domain. With this understanding and assuming meaningful code generators, the impact of changing the content of a model is more coherent than to directly modify source code.

On the other hand, the initial cost for using models constructively is high: Designing a modeling language requires a detailed understanding of the problem domain (problem space) the language targets at. This understanding may be split among different stakeholders, which need to be intensively consulted to develop the modeling language as suitable as possible. Code generator or interpreter developers must additionally bring expertise in solving the problems described the modeling language. In summary, this requires a detailed understanding of the following aspects:

1. The parts of the problem domain that can be described using the modeling language (problem space)
2. The modeling language itself (model space)
3. How the models relate to the parts of the problem domain they describe (relation between model and problem, i.e., explicit description)

4. The solution space for solving the subjects described by models conforming to the language (solution space)
5. How to solve the problem described by a model using the provided solution space (relation between model and solution, i.e., realization of implicit description)

Thus, for applying constructive MBSE successfully, expertise in the solution space as well as the problem domain and knowledge about solving problems described by models is required prior to code generator or interpreter development. As models may describe nontrivial circumstances in complex domains, code generator or interpreter development may require highly skilled personnel.

On the contrary, in traditional software engineering projects, the model space and, therefore, the relations between models and problems as well as solutions do not exist. In such projects, it is easier to adapt to changing requirements (changes in the problem space) as these only require to adapt the solution. In MBSE projects, where models are used constructively, changes of the problem space may require costly adaptations of the model space as well as the relations between the model space and the other two spaces. With the increasing shift to agile development methodologies, dynamically changing requirements become a standard.

3 Failures of Model-Based Software Engineering

Even though MBSE promises to reduce complexity and decrease development time, there are pitfalls and challenges that may lead MSBE projects to fail. In the remainder of this section, we summarize these pitfalls and challenges.

Creating and Maintaining Code Generators An integral part of MBSE are code generators for translating models to source code. Current tool support (cf. [15, 36]) enables rapid and easy code generator development for arbitrary models. This might be true for relatively simple code generators that do not analyze the input models to adapt the generated source code. However, this situation is different when code generators become complex. This is often the case when complete software systems are generated, for example, [28]. In this situation, a code generator has to be regarded as a complex software system on its own. In some cases, it may also be a product line (cf. [42]) or needs to process modeling language product lines [36]. This demands for additional effort to develop and maintain the code generator because for every change in the input language or the generated source code, the code generator has to be adapted.

Design and Development of DSLs Each code generator relies on a model, which is an instance of a particular (or multiple) DSL. Each DSL is specifically tailored to a particular domain and potentially has to be adapted when reusing it for a different domain or a different scenario. However, designing and developing a DSL is, in general, a complex and time-consuming task, which is only partially supported by design guidelines [25]. This additional development effort has to be invested in order to effectively and efficiently employ MBSE. The state of the art in software language engineering only partially supports reusing parts of preexisting DSLs and corresponding tooling [36]. While language modularization scenarios for syntax definitions are widely covered by existing language workbenches [36], tool composition for processing models (e.g., code generator composition) seems to be still an open problem [37].

Modeling Needs Domain Knowledge In general, modeling requires a deep understanding of the modeling language and, more importantly, the domain itself. A modeler has to understand the requirements and demands of the domain and must be able to transform them into appropriate models (cf. Sect. 2.3). This task is, generally, (a) more complex than programming (sequential thinking versus specification-oriented thinking) and it involves (b) creating concise and possibly underspecified models, which demands an unconventional thinking process.

Compilers Are Highly Optimized; DSL Code Generators Just Do Their Work For GPLs, compilers are highly optimized to particular needs, for example, runtime performance. In contrast, models are processed by code generators or interpreters (cf. Sect. 2.2). Both are generally not highly optimized. Hence, for particular types of software systems, for example, high-performance algorithms, a MBSE approach fails if it does not make use of sophisticated adaptation mechanisms. Such mechanisms allow to adapt either the code generator, interpreter, or the generated source code to perform optimization tasks or handle underspecification. However, even if adaption is possible, the adaption process may still be more expensive (in terms of, e.g., development time, maintainability, etc.) than writing efficient code from scratch.

Project Setting Has to Fit MBSE A typical reason for MBSE project failure is rooted in the nature of the project itself. MBSE software projects are highly effective for software projects with repetitive and similar tasks. For example, imagine a software project that defines a REST interface for a particular domain model. In this project, the technology stack and the definition of a REST interface are clearly similar. Hence, suitable code generators can speed up development time by generating the REST interface from the domain model. However, for software projects without or with only little repetitive tasks—where every part of the source code is very individual—MBSE fails because it demands a code generator and a suitable DSL for each particular concern.

Model Tooling Is Not That Elaborated as for Programming (IDEs) The lack of tooling for model and code generator creation is a major challenge of MBSE. Even though there are language workbenches (cf. [15, 36]) that support generation or manual implementation of integrated development environments (IDEs) with syntax highlighting and auto completion, the resulting tools are not as supportive as well-established IDEs for GPLs. Furthermore, the tool support has to be maintained whenever the DSL evolves. A similar situation is also present in the development of code generators, where suitable adaptation concerns have to be addressed by additional infrastructures (cf. [21]).

MBSE and Agility Do Not Yet Work Well Together Agile software development has shown its effectiveness as a method to tackle complexity and ambiguity of software projects and to reduce the time to deliver software products. However, MBSE and agile software development methods are still not yet coalesced and demand for a combination of plan-driven and agile methods (cf. [27]). More importantly, the crucial role of model quality and tooling for creating, managing, and refactoring models is not yet resolved (cf. [8, 26]).

From the aforementioned pitfalls and failures of MBSE and from current research, the main questions to be answered are: *What kind of software systems can effectively be supported by MBSE? What needs to be improved to make MBSE suitable?* Clearly, software language engineering and code generator engineering needs to improve. This also holds for tooling supporting language reuse and composition as well as code generator and interpreter composition and creation. Hence, in the following, we present success stories of MBSE as well as the use cases' motivations and achievements with respect to MBSE. These examples are helpful to understand when MBSE is appropriate and why it is successful.

4 Where Model-Based Software Engineering Is Successful

Example domains where MBSE has proven successful are cyber-physical systems (CPS) [2] and component-based software engineering (CBSE) [38]. In these contexts, MBSE has been adopted by means of architecture description languages (ADLs) [35]. ADLs are special modeling languages providing syntax to describe the structure of a system under development. Academia and industry produced over 120 ADLs [29] in context of different domains such as automotive [13], avionics [16], consumer electronics [53], or robotics [46]. In context of ADLs, code generators take models describing the structure of a system under development as input and produce a runtime infrastructure for executing the system. Such models typically abstract away component implementation details and focus on the system's structure, that is, the system's components and their interconnections. The implementation of each individual component thus typically has to be implemented by hand. Some ADLs additionally enable to describe component behavior implementations. Where this possibility is available, it is even possible to generate complete software systems. Examples for graphical and textual CBSE-related DSLs are integrated into the AutoFocus [3], MontiArc [6], and Simulink [47] frameworks.

Other example domains successfully employing MBSE that use model interpretation are the domains of software build automation and web development. Makefiles, for example, are executed by command line interpreters. Web browsers interpret HTML models as text layout definitions and interpret JavaScript programs, for instance, to execute a reaction in response to user interaction events.

The neuroscience domain has successfully produced the NestML, which is a DSL used by neuroscientists for creating neuron models in a precise and concrete syntax familiar to domain experts [40]. A code generator translates NestML models into

C++ code that can be dynamically loaded into NEST [20], a simulator for networks of spiking point neurons. Before integrating NestML, domain experts created new neuron models for NEST by copy-and-paste from existing models. As neuroscientists are no programming experts, source code redundancy, suboptimal performance, improper documentation, and reduced maintainability are the consequences [40]. Investigations revealed cases where two different copy-and-paste models shared more than 90% of their contents [40]. The modeling language and its tools were well received by domain experts [40]. An evaluation revealing the increase in code reuse, performance, and documentation as well as the decrease in redundancy is currently not available.

MBSE has also been successfully applied in context of a customs information system to describe document verification rules and configuration files [18]. Such verification rules describe constraints that relate different fields of XML documents. A compiler translates verification rule models to Java byte code. A requirement on the verification rule DSL was that its models should be similar to rules expressed in natural language [18]. This indicates that verification rules are modeled by nontechnical users.

Another case study reports on successful application of MBSE in context of service robotics applications [1]. The goal is to achieve separation of concerns and to ease development of complex service robotics applications by applying models and code generators. Domain experts model the environment a robot operates in, robot and world abilities, as well as tasks and goals for the robot to achieve. The goal is to enable nontechnical domain experts to model the context of a service robotics application. From such models, generators produce scenario-specific source code. Robotics experts then complement the generated code with handcrafted implementations for the world and robot abilities at well-defined places. Thus, models capture the requirements of a particular instance in the service robotics domain (robot environment, robot/world abilities), and robotics experts complement the application with their robotics domain expertise. This ultimately reduces the costs for developing service robotics applications and enables separation of concerns.

JavaSM is a DSL that integrates state machines into Java to develop a command-and-control simulator [4]. The case study also relies on CBSE and product line architecture methodologies. The DSL is used for defining and refining state machines that specify component behavior. State machine-based implementations are frequently used in this case study. The motivation for developing the JavaSM is that state machine encodings in pure Java are extremely complex and thus hard to maintain and understand [4]. The achievement is a complexity reduction of component specifications: The case study's result is a more flexible way of implementing state machine-based simulators than is possible with pure Java implementations [4]. Hence, the case study is an example of where a DSL eases software development in a particular use case and hence increases productivity of developers.

Risla is a DSL for describing products developed in the financial engineering domain [52]. Financial engineering experts use Risla to model new products. The case study's motivation is to decrease the time needed for introducing new types of products and to ensure newly developed products are correctly implemented as intended [52]. The complexity of introducing a new product seems to be based on the case study's underlying software system, which is implemented in COBOL. The second motivation arises since it is only hardly, if even, possible to ensure a software engineer correctly implements the instructions given by a financial engineer. The goal is to employ models and code generators such that (1) new products can be added easily and (2) potential information loss during communication of domain experts and software engineers is avoided. This ultimately leads to a lower time to market for new products and eliminates a source for introducing incorrect behavior.

Other examples for well-known successful DSLs commonly used in (computer) science domains include, among many others, SQL for specifying database queries, LaTeXfor writing documents, and MATLAB [33] for numerical computing and embedded systems. Other domains where DSLs have been used include artificial intelligence [34], graphics [14], model checking [23], operating systems [41], various protocols [7, 49], and video device drivers [50]. Most of these DSLs target at increasing productivity of software developers and ensuring program correctness.

5 Can We Draw Conclusions?

While MBSE has its drawbacks and failures, it still keeps its promises in some domains and software projects. In particular, summarizing the reasons for successful MBSE, we obtain the following indicators:

1. If a company is aware of its software needs to be available in many different versions and continuously evolves, and development continues for a long time, then MBSE is helpful. Models then act as abstract specifications for individual features as well as the overall architecture. Production code and test code generators are in place to decouple the application from the technology stack and to embrace variant management. This is achieved by using models as composable units (features) selected with respect to the needs of each individual product.
2. Domain experts are not necessarily classical software developers (programmers) but may also be nontechnical stakeholders without programming expertise. Such experts may prefer to use a DSL or a graphical modeling language rather than a GPL. In this case, models enable domain experts and users to actively participate in system development. Domain experts provide domain models to develop the system and users use models to configure it.
3. If a system is not only composed of software but also of complex hardware, then there is no alternative to using models for describing the overall system. Physical models are used by any engineering discipline, to understand a system under development before building it. If the system's software part is also to be understood early, then integrated physical and software models are helpful. The CPS and Industry 4.0 initiatives are shifting towards this direction.

4. If a system is grounded on a well-defined theory such as physics, chemistry, or biology, then scientists developing concrete models of their system under examination prefer to use models with syntax similar to the underlying mathematical theory. Such models are preferably used for simulation of real-world phenomena such as galaxies, climate, brain, chemical molecules, or physical particles.

In reverse conclusion, the overhead for initializing MBSE development processes often does not pay off compared to the benefits obtained from reusing models and generators in a modular way. This is the case, for instance, if a developing company is ensure whether an enhanced variant of an already existing software product needs to be delivered in a future project. This is often the case in purely software-based systems as, for example, enterprise information systems and software developed in web domains.

However, the use of models has still not yet been explored in all potentially possible domains. There are promising examples, such as nontechnical users writing Excel formulas without explicitly noticing they are actually programming. Supporting end users to program in a restricted and controlled fashion could be supported by appropriate modeling languages.

Like many other new and innovative technologies, MBSE has promised a lot and has not kept all of its promises. Fortunately, a lot of ideas and techniques from MBSE can be used for implicit modeling using GPLs. The encoding of state machines using the state design pattern [19] is such an example.

References

1. Adam, K., Butting, A., Heim, R., Kautz, O., Rumpe, B., Wortmann, A.: Model-driven separation of concerns for service robotics. In: International Workshop on Domain-Specific Modeling (DSM'16). ACM, New York (2016)
2. Alur, R.: Principles of Cyber-Physical Systems. The MIT Press, Cambridge (2015)
3. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: tooling concepts for seamless, model-based development of embedded systems. In: Joint Proceedings of ACES-MB 2015 - Model-Based Architecting of Cyber-Physical and Embedded Systems and WUCOR 2015 - UML Consistency Rules (2015)
4. Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: a case study. ACM Trans. Softw. Eng. Methodol. **11**(2), 191–214 (2002)
5. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synth. Lect. Softw. Eng. **1**(1), 1–182 (2012)
6. Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., Wortmann, A.: Systematic language extension mechanisms for the MontiArc architecture description language. In: Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017, pp. 53–70. Springer International Publishing, Cham (2017)
7. Chandra, S., Richards, B., Larus, J.R.: Teapot: language support for writing memory coherence protocols. In: Conference on Programming Language Design and Implementation. ACM, New York (1996)
8. Chaudron, M.R., Heijstek, W., Nugroho, A.: How effective is UML modeling? Softw. Syst. Model. **11**(4) (2012)

9. Chen, P.P.S.: The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst. **1**(1), 166–192 (1976)
10. Cheng, B., Eder, K., Gogolla, M., Grunske, L., Litoiu, M., Müller, H., Pelliccione, P., Perini, A., Qureshi, N., Rumpe, B., Schneider, D., Trollmann, F., Villegas, N.: Using models at runtime to address assurance for self-adaptive systems. In: Models@run.time. Lecture Notes in Computer Science, vol. 8378, pp. 101–136. Springer, Berlin (2014)
11. Cheng, B.H.C., Combemale, B., France, R.B., Jézéquel, J.M., Rumpe, B.: Globalizing domain-specific languages (Dagstuhl Seminar 14412). Dagstuhl Rep. **4**(10) (2015)
12. Combemale, B., France, R., Jézéquel, J.M., Rumpe, B., Steel, J., Vojtisek, D.: Engineering Modeling Languages: Turning Domain Knowledge into Tools. Innovations in Software Engineering and Software Development Series. Chapman & Hall/CRC, London/Boca Raton (2016)
13. Debruyne, V., Simonot-Lion, F., Trinquet, Y.: EAST-ADL - an architecture description language. In: Architecture Description Languages. Springer, New York (2005)
14. Elliott, C.: Modeling interactive 3D and multimedia animation with an embedded language. In: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997. USENIX Association, Berkeley (1997)
15. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches. Comput. Lang. Syst. Struct. **44**, 24–47 (2015)
16. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional, Reading (2012)
17. Fieber, F., Huhn, M., Rumpe, B.: Modellqualität als Indikator für Softwarequalität: eine Taxonomie. Informatik-Spektrum **31**(5), 408–424 (2008)
18. Freudenthal, M.: Domain-specific languages in a customs information system. IEEE Softw. **27**(2), 65–71 (2010)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
20. Gewaltig, M.O.: NEST (NEural Simulation Tool). Scholarpedia **2**(4), 1430 (2007)
21. Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., Navarro Perez, A., Plotnikov, D., Reiß, D., Roth, A., Rumpe, B., Schindler, M., Wortmann, A.: Integration of handwritten and generated object-oriented code. In: Model-Driven Engineering and Software Development. Communications in Computer and Information Science, vol. 580. Springer, New York (2015)
22. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Comput. **37**(10), 64–72 (2004)
23. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)
24. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)
25. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S.: Design guidelines for domain specific languages. In: Domain-Specific Modeling Workshop (DSM'09), Techreport B-108. Helsinki School of Economics (2009)
26. Kent, S.: Model Driven Engineering. In: Integrated Formal Methods. Lecture Notes in Computer Science, vol. 2335. Springer, Berlin (2002)
27. Kulkarni, V., Barat, S., Ramteerthkar, U.: Early experience with agile methodology in a model-driven approach. In: 14th International Conference on Model Driven Engineering Languages and Systems. MODELS'11. Springer, Berlin (2011)
28. Look, M.: Unterstützung modellgetriebener, agiler Entwicklung mehrbenutzerfähiger, ubiquitärer Enterprise Applikationen durch Generatoren. Ph.D. thesis, RWTH Aachen University, Aachen (2017)

29. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6), 869–891 (2013)
30. Maoz, S., Ringert, J.O., Rumpe, B.: A manifesto for semantic model differencing. In: Proceedings International Workshop on Models and Evolution (ME'10). Lecture Notes in Computer Science, vol. 6627, pp. 194–203. Springer, Berlin (2010)
31. Maoz, S., Ringert, J.O., Rumpe, B.: ADDiff: semantic differencing for activity diagrams. In: Conference on Foundations of Software Engineering (ESEC/FSE '11), pp. 179–189. ACM, New York (2011)
32. Maoz, S., Ringert, J.O., Rumpe, B.: Semantically configurable consistency analysis for class and object diagrams. In: Conference on Model Driven Engineering Languages and Systems (MODELS'11). Lecture Notes in Computer Science, vol. 6981, pp. 153–167. Springer, Berlin (2011)
33. Matlab Homepage (2017). https://de.mathworks.com/products/matlab.html. Accessed 09 May 2017
34. Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL - the planning domain definition language. Tech. Rep. TR-98-003, Yale Center for Computational Vision and Control (1998)
35. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
36. Méndez-Acuña, D., Galindo Duarte, J.A., Degueule, T., Combemale, B., Baudry, B.: Leveraging software product lines engineering in the development of external DSLs: a systematic literature review. Comput. Lang. Syst. Struct. **46**, 206–235 (2016)
37. Mir Seyed Nazari, P., Roth, A., Rumpe, B.: An extended symbol table infrastructure to manage the composition of output-specific generator information. In: Modellierung 2016 Conference, vol. 254. Bonner Köllen Verlag, Bonn (2016)
38. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee (1969)
39. Object Management Group (2017). http://www.omg.org. Accessed 09 May 2017
40. Plotnikov, D., Blundell, I., Ippen, T., Eppler, J.M., Morrison, A., Rumpe, B.: NESTML: a modeling language for spiking neurons. In: Modellierung 2016 Conference. LNI, vol. 254. Bonner Köllen Verlag, Bonn (2016)
41. Pu, C., Black, A., Cowan, C., Walpole, J., Consel, C.: Microlanguages for operating system specialization. In: Workshop on Domain-Specific Languages (1997)
42. Roth, A., Rumpe, B.: Towards product lining model-driven development code generators. In: Model-Driven Engineering and Software Development Conference (MODELSWARD '15). SciTePress, Setúbal (2015)
43. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer International, Cham (2016)
44. Rumpe, B.: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer International, Cham (2017)
45. Schindler, M.: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, Herzogenrath (2012)
46. Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: communication patterns as key for a robotics component model. In: Introduction to Modern Robotics. iConcept Press, Kowloon (2011)
47. Simulink Homepage (2017). https://de.mathworks.com/products/simulink.html. Accessed 09 May 2017
48. Systems Modeling Language (2017). http://www.omgsysml.org/. Accessed 09 May 2017
49. Thibault, S., Consel, C., Muller, G.: Safe and efficient active network programming. In: Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No.98CB36281) (1998)
50. Thibault, S.A., Marlet, R., Consel, C.: Domain-specific languages: from design to implementation application to video device drivers generation. IEEE Trans. Softw. Eng. **25**(3), 363–377 (1999)

51. Unified Modeling Language (2017). http://www.omg.org/spec/UML/. Accessed 09 May 2017
52. van Deursen, A., Klint, P.: Little languages: little maintenance. J. Softw. Maint. **10**(2), 75–92 (1998)
53. Van Ommering, R., Van Der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)