Generative versus Interpretive Model-Driven Development: Moving Past 'It Depends'

Michiel Overeem[1(✉)], Slinger Jansen[2], and Sven Fortuin[2]

[1] Department of Architecture and Innovation, AFAS Software, Leusden, The Netherlands
m.overeem@afas.nl

[2] Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands
{slinger.jansen,s.e.fortuin}@uu.nl

Abstract. Model-driven development practices are used to improve software quality and developer productivity. However, the design and implementation of an environment with which software can be produced from models is not an easy task. One part of such an environment is the model execution approach: how is the model processed and translated into running software? Experts state that code generation and model interpretation are functionally equivalent. However, a survey that we conducted among several organizations shows that there is a lack of knowledge and guidance in designing the execution approach. In this article we present the results of a literature study on the advantages of both interpretation and generation. We also show, using a case study, how these results can be utilized in the design decisions. Finally, a decision support framework is proposed that can provide the guidance and knowledge for the development of a model-driven engineering environment.

Keywords: Model-driven development · Model-driven architecture
Software architecture · Code generation
Run-time model interpretation · Decision support

1 Introduction

Model-driven development (MDD) is used by software producing organizations (SPOs) to improve software quality and developer productivity. According to

This work is a result of the AMUSE project. See https://amuse-project.org for more information. An earlier version of this work was published as Overeem and Jansen [1]. This article adds the motivating survey that we have conducted among sixteen organizations. The literature study is extended with literature published since the original study. The case study is expanded with more in-depth observations on the decision making process.

Díaz et al. [2] these improvements in quality and productivity are achieved because a well designed model raises the abstraction level of the software development process. The abstracted model allows for an expressiveness that can be more concise than general-purpose programming languages. Domain-specific modeling improves that even further by catering the model to a certain domain. The expressiveness causes both the increase of productivity (more can be done with less) and the quality (there will be fewer mistakes, because there is a smaller model). The models can be used in different manners, Brown [3] shows a modeling spectrum with, among others, roundtrip engineering, model-centric, and model only. We are especially interested in the model-centric approach: the model is the source of truth and the application follows from the model. The model-centric approach is implemented in Model Driven Engineering Environments (MDEE), an environment that is similar to an Integrated Development Environment (IDE) used for software development. Modelers create models using modeling languages in a specific modeling environment, just as developers write software in their IDE. These models are translated according to well defined semantics, into an application. Together these components (from modeling environment up to and including the application) form the MDEE (visualized in Fig. 1). The translation process that reads the model and produces an application is defined as the *model execution approach*, and implemented in the model execution engine.

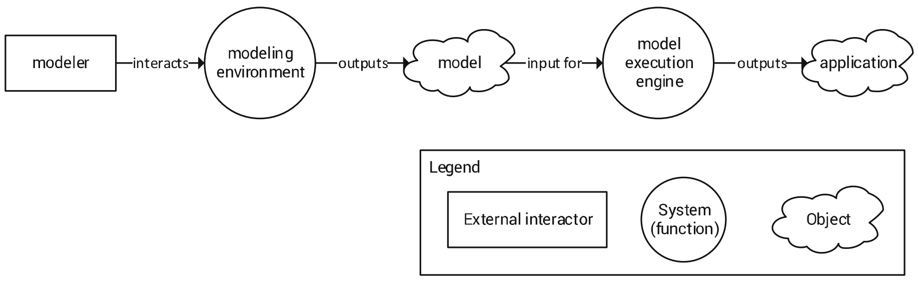

Fig. 1. A model-driven engineering environment enables a modeler to create a model in a *modeling environment*. The model is subsequently translated by the *model execution engine* (using a *model execution approach*) into an application.

Our experiences are that the development of a MDEE is by no means an easy task. The initial investment is large, because there are many technical challenges. One of these technical challenges that is of particular interest to us, is the design and development of the *model execution approach*. SPOs can choose for code generation, run-time interpretation, or a hybrid form that combines both approaches (Fig. 2). As in every design challenge, there are numerous decisions to make (with their specific trade-offs) that influence the overall quality of the MDEE.

Just like any other (architectural) design question, the design questions for the model execution approach can be answered with "it depends". In this article we show that the design depends on desired quality characteristics and the context of the MDEE. Moreover, we show how SPOs can take these characteristics into account. It might be regarded as an implementation detail, but the model execution approach, like any other component in the system has its influence on the quality characteristics (such as run-time behavior and maintainability) of the whole system. As in any system that consists of multiple components working together, the model execution approach should not be designed individually (i.e., not be out of the context of the MDEE). The influence of the model execution approach is similar to, for example, the influence of a specific database on the quality of a data intensive system. While users may not see a difference in functionality between two different databases, the quality of the system is affected by it, for example, in terms of performance, stability, and availability. SPOs can deliver the same functionality, whether they choose code generation or run-time interpretation, but the quality of the MDEE will differ significantly.

The main research question of this article is *How can SPOs make an informed decision between a generative or interpretive model execution approach?*. In Sect. 2 we explain the different model execution approaches in more depth, and discuss the work already done in this area. We motivate our research question in Sect. 3 by presenting the results of a survey among SPOs applying model-driven development. This survey shows that there is no "one size fits all" solution. It also shows that many SPOs do not have a clear rationale for the model execution approach that is used. Therefore, decision support and clear guidance is necessary to improve the overall design and implementation of MDEEs. Section 4 discusses the results of the literature study that we have performed on the advantages and disadvantages of the generative and interpretive approach. There are many hybrid model execution approaches that combine the generative and interpretive approach. We show the preference for the two pure approaches in terms of percentages. These percentages can be used by the SPO to find the right balance in designing their own hybrid model execution approach. Section 5 describes a case study, in which we observe the design of a fitting model execution approach. We conclude that the design of a fitting model execution approach is not detached of the overall design of the MDEE. We reflect on the case study and our observations in Sect. 6. We observed three general areas of design decisions that influence the model execution approach, and we present a design support framework based on the case study. Finally, Sects. 7 and 8 evaluates and discusses the study, and presents our conclusion respectively.

2 Context and Related Work

There are several model execution approaches, many of them are a hybrid form of the two pure approaches. We discuss the two pure approaches, and describe two groups of hybrid approaches, shown in Fig. 2. The first pure model execution approach is *code generation*. During code generation a model is parsed,

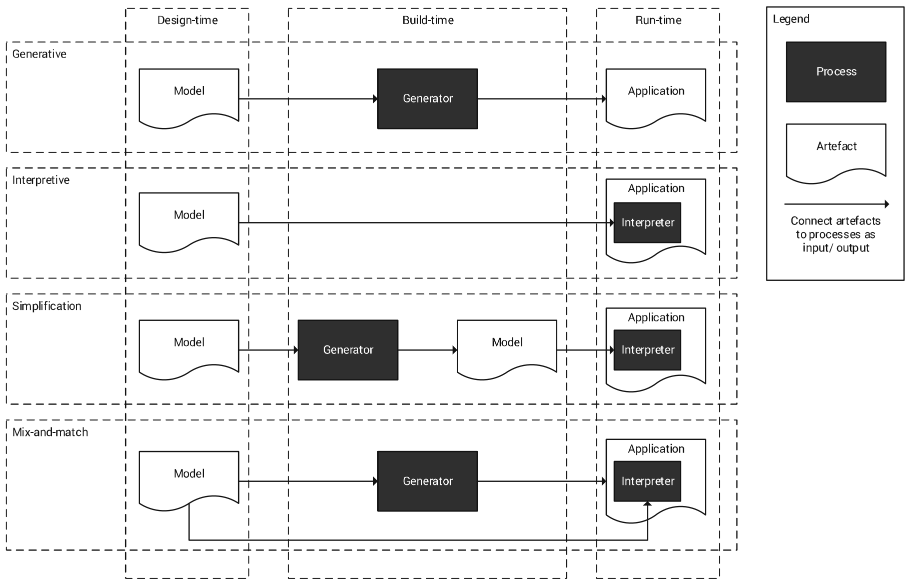

Fig. 2. The four main types of execution approaches are generation, interpretation, simplification, and mix-and-match. The darker boxes show the execution process. With the two hybrid approaches, the execution process can be split up and divided between build-time and run-time. The model is created at design-time, but is used at build-time and/or run-time.

interpreted and transformed into source code. The generated source code generally results in running software. This approach is not exclusive to MDD, and is formalized and defined by Czarnecki and Eisenecker [4] as *Generative Programming*. According to their definition, it is a paradigm based on modeling facilities to automatically manufacture customized and optimized intermediate and/or end-products. Applying generative programming within MDD results in generative MDD. Although nothing in the definition states that the output can not be changed manually before the final software product is delivered, we only regard *full* code generation that does not need manual changing the generated code. This does not imply that every part should be generated; the generated code can be combined with frameworks or base libraries, as pointed out by Kelly and Tolvanen [5].

The second pure model execution approach is *run-time interpretation*, or interpretive MDD. The idea is similar to code generation, but the timing is different: the parsing and interpretation of the model are done at run-time. There is no need to first generate source code, the running software executes its functions directly based on the model. In this case the model execution approach becomes part of the application, the application interprets the model before offering functionality based on the model. Further manual coding is not possible with this approach, because there is no time to intervene in the execution of the software. However, as we see in Sect. 3 it is possible to combine custom code with an interpreter. The model needs to be deployed along with the running software, while in the generative approach, the model is not part of the running software.

These two approaches form the extremes of the execution spectrum, and many hybrid forms are possible. We see two groups of hybrid approaches. The first group is *simplification*: a model is transformed into a second model before deploying it for run-time interpretation. In this approach there is both a generation step and an interpretation step, instead of generating source code. The generation step transforms the model into a second model that can be interpreted at run-time. This can be achieved by transforming high-level concepts to low-level concepts, or by transforming into a model with fewer constructs. The results of this approach are manifold: (1) The interpreter is easier to develop and better maintainable, because it has to support fewer constructs. (2) The translation is less complex, so the interpreter is faster. And finally, (3) the interpreter becomes more reusable, because there can be many different models that can be transformed into the intermediate model. This approach is also used by programming languages that compile into an intermediate language that is in turn interpreted by a runtime environment, such as the approach Meijler et al. [6] discuss. They generate Java source code, but use a customized class loader that acts as a run-time interpreter.

The second group of hybrid approaches is a *match-and-mix* approach: some parts of the platform use code generation, while others use interpretation. This approach can be used both from an architectural perspective as well as from a model perspective. The MDEE could use a different approach in different components, for instance the user interface could be interpreted, while the database access layer is generated. Different approaches could also be chosen based on model dynamics, where the more stable parts can be generated into source code and the more dynamic parts are interpreted.

Figure 2 shows the four described approaches, marking out the time at which the execution takes place. In the generative approach, the execution is done at build-time, as opposed to the interpretive approach in which the execution takes place at run-time. Both the simplification and mix-and-match approach show that they have part of the execution at build-time, and part at run-time. This makes them flexible, because SPOs can decide how much happens at what time. These hybrid approaches can also be combined, the mix-and-match approach can combine the interpretive, generative, and simplification approach into a single encompassing model execution approach.

There is some work done on the challenge of designing a fitting model execution approach discussed in this article. A multi-criteria analysis of the different approaches is performed by Batouta et al. [7] with as goal the support of the decision-making. Their analysis results in a decisive statement about the best approach (based on their list of ten criteria). However, they do not take the context of the MDEE into account. Fabry et al. [8] address a number of advantages regarding the different model execution approaches, but they do not give any support for the decision-making. Zhu et al. [9] researches the decision-making within

MDD applied to game development, however, he only looks at other architectural decisions than the model execution approach within a MDEE. Code generators and the interaction with developers is researched by Guana and Stroulia [10], only without making a comparison with the interpretive approach. All of the mentioned work is incorporated in the literature study in Sect. 4.

The design of software and their architectures is a thoroughly researched topic. Capilla et al. [11] show how design decisions play a role in software architecture, and that it is important to capture them. Jansen and Bosch [12] define "software architecture as the composition of a set of architectural design decisions", and formalize this in the Archium approach which is further extended in Ven et al. [13]. Svahnberg et al. [14] present a decision process that, based on desired quality attributes, supports a SPO in finding the architecture variant that shows the most potential. We combine the definition of software architecture as a set of design decisions with the approach to support a decision with quality attributes, and apply this to MDD. Because of this we are able to uncover the rationale of either a generative or interpretive approach, and support SPOs in their design process.

3 How SPOs Design and Develop MDEEs

We interviewed twenty-two product experts of sixteen different SPOs that develop MDEEs. All of the experts had either five or more years experience with the product or were working with the product since its start. They served in different roles at the time: twelve of them as chief executive, the others in different roles such as lead developer, business developer, and sales manager. These experts were asked questions on the design and implementation of their company's MDEE. The SPOs were identified by an Internet search, exploiting our network, and asking interviewed product experts.

We identified 36 qualifiable case companies with representatives in Belgium, The Netherlands, or Luxembourg. For sixteen companies we found experts that were willing and able to cooperate in our research[1]. The companies differ in size (ranging from ten employees to thousands of employees), in market (some operate only in The Netherlands, while others operate worldwide), and maturity (some MDEEs are almost twenty years old, while others only two years). After we processed the answers, every expert had the opportunity to correct any mistaken interpretations. The answers are summarized in Table 1.

The first topic of interest is the target users of the MDEE and its modeling language. We asked the experts what the target group of users for the MDEE are, and what kind of expertise they expect from them. Their answers resulted in four categories of users:

[1] Some needed to be excluded due to confidentially issues or the lack of (technical) knowledge.

Table 1. Anonymized results of the survey among SPOs. The target users and the model execution approach are shown, a long with the company size (in terms of number of employees) and maturity (in terms of number of development years). The cells marked with an * identify MDEEs that support two distinct web platforms.

	SPO_1	SPO_2	SPO_3	SPO_4	SPO_5	SPO_6	SPO_7	SPO_8	SPO_9	SPO_{10}	SPO_{11}	SPO_{12}	SPO_{13}	SPO_{14}	SPO_{15}	SPO_{16}	
Company size																	
0-50 employees	●	●	●	●			●					●		●	●	●	56%
100-500 employees						●		●	●	●			●				31%
+500 employees					●						●						13%
Development years																	
0-5			●						●			●					19%
6-15	●	●		●						●					●	●	37%
+15					●	●	●	●			●		●	●			44%
Target platforms																	
Web	●	●	●	●	●	●*	●	●*	●	●	●	●	●	●*	●	●	100%
Desktop				●		●							●				19%
Mobile													●				6%
Target users																	
Laymen	●			●					●			●			●		32%
Technical business users		●	●		●			●		●	●		●	●		●	56%
SQL experts				●													6%
Developers	●	●				●	●			●						●	37%
Model execution approach																	
Interpretation	●	●	●	●		●	●			●			●	●	●	●	69%
Generation	●	●				●		●	●		●	●	●			●	50%
Simplification					●				●								13%
Match-and-mix	●	●				●			●				●			●	38%

- **Laymen** are people without any technical knowledge.
- **Technical business users** are those that have some knowledge of software development, but are no developers. They are expected to have knowledge about software concepts such as data models, and data types. An informal description would be people more knowledgeable than layman, but less knowledgeable than developers.
- **SQL experts** are a specific set of users that are able to write SQL queries. They are not able to write software in other programming languages. This specific category was added after the review with SPO_4, because the category **developers** did not match their target description.
- **Developers** are those users that are able to write software in a programming language. MDEEs that target developers expect them to be familiar with IDEs and other programming concepts.

A third of the SPOs specifically target laymen, while the others require some form of technical knowledge of their users. There is no correlation found between the model execution approach and the targeting of laymen. In the case study we conducted (see Sect. 5) we also observe the design of a MDEE that targets laymen. A third of the SPOs that target technical business users also target developers, their MDEEs support custom programming, because the model is not able to express al required functionality. The six SPOs that target developers all use an interpretive approach, four of them also use code generation.

Five SPOs (SPO_2, SPO_6, SPO_9, SPO_{13}, and SPO_{16}) state that a reason for their model execution approach is a certain required build-time behavior. As an example, the expert of SPO_{16} states "*You can't generate code again in an end application that is already generated. To allow workflow modeling in the end application, we were forced to make use of an interpretative solution.*". All of the five mentioned SPOs explain that users are able to change the model, and expect that their changes are (near) instantly applied and visible in the application. Four of them use run-time interpretation, while the other one uses a simplification approach. The SPOs that use a generative approach did not mention such a requirement for build-time behavior.

All of the SPOs target a web platform, meaning that they support at least back-end and front-end applications. Three of the SPOs, however, support two different back-end platforms, one also supports mobile applications, and two others also support native desktop applications. Effectively, we can conclude that all SPOs support multiple platforms. The interpretive approach is motivated three times by the advantage of platform independence, or portability.

We have found little reasoning behind the implemented approaches, one expert even stated "*We just had to go with one of the two.*". An expert of SPO_6 refers to an advantage in portability for interpretation, a correlation that we will see again in Sect. 4: "*By interpreting the UI and generating the remaining parts of the application, we are able to share models between different platforms.*". A reference to resource utilization is made by an expert of SPO_{14}: "*We don't want to regenerate an entire database every time the model changes, because this can potentially cause a lot of problems with data migration.*". The interviews show that all approaches are used, and nearly half of them use a hybrid form. This supports our claim that the model execution approach depends on many factors and is context specific. We cannot give a simple answer such as "web platforms should use an interpreter", Table 1 shows that other approaches are used for web platforms as well. Like Capilla et al. [11] we believe that it is important for SPOs to document the rationale behind important architectural decisions. In the next Section we will show that the model execution approach influences the quality of the MDEE, and that it is important to capture the rationale of the design.

4 Quality Characteristics of Model Execution Approaches

We started the literature study by executing a literature review on the advantages and disadvantages of both code generation and run-time interpretation. The literature review was done with the snowballing approach as described by Wohlin [15]. The snowballing approach uses references between articles as a means to discover other relevant literature. The first step is to select a start set from which the references can be followed. This approach was chosen because the research areas to be covered in this review are broad. We expected literature from the MDD field as well as Domain-Specific Language engineering and compiler design. The second reason was that the literature that we had found in earlier explorations never mentioned the advantages or disadvantages directly, but they where often hidden in implementation details.

Our start set was created by earlier informal explorations with the Google Scholar engine, using "interpretation versus code generation" and "interpretation vs. code generation" as keywords. We selected five articles as the start set: van Deursen et al. [16], Meijler et al. [6], Mernik et al. [17], Tanković [18], and Voelter [19]. These papers represent the different research areas and have a broad research question, resulting in many references (both backwards and forwards). With this start set we executed several steps, following both backward and forward references. The found literature was included when it mentioned advantages or disadvantages on model execution approaches, and we ended up with 35 studies.

The literature was classified using the ISO standard 25010:2011 for software product quality [20]. This standard is used to asses the quality of software systems, and matches our intent to asses the quality of MDEEs. The ISO standard consists of eight categories with 31 characteristics. We found evidence for differences in quality fulfillment for five out of these eight categories, summarized in Table 2. The summary of all the found evidence is presented in Table 3. There was no evidence found for the categories *functional suitability*, *usability*, and *reliability*. The first two categories match the statement of Stahl et al. [21] "code generation and model interpretation are functionally equivalent". For category *reliability* no evidence was found as well, which was expected. Reliability is the degree to which the system performs its functions under certain conditions. We assume the generative and interpretive approach to be functionally equivalent, and in both approaches it is possible to build a reliable functioning system.

Table 3 presents every mention of an advantage or disadvantage in relation to its source and the quality characteristics. A G stands for a preference of generation over interpretation, while an I stands for the opposite. When we encountered statements on the two approaches without a preference, we marked the corresponding cell with both G and I. The total number of preferences are used to calculate the percentage of the two alternatives with respect to the quality characteristic. The evidence we found is presented in relation to the generative and interpretive approach. This does not mean that the hybrid approaches are not mentioned by authors (as discussed in Sect. 2). However, the advantages and disadvantages we found were always in terms of the generative and interpretive aspects of an approach.

Table 2. The categories and characteristics from the software product quality model in ISO standard 25010:2011. For the emphasized items we found evidence of a preference for either code generation or model interpretation.

Category	Characteristics
Functional suitability	Functional completeness, Functional correctness, Functional appropriateness
Performance efficiency	*Time behaviour*, *Resource utilization*, Capacity
Compatibility	*Co-existence*, *Interoperability*
Usability	Appropriateness recognizability, Learnability, Operability, User error protection, User interface aesthetics, Accessibility
Reliability	Maturity, Availability, Fault tolerance, Recoverability
Security	*Confidentiality*, Integrity, Non-repudiation, Accountability, Authenticity
Maintainability	*Modularity*, Reusability, *Analysability*, *Modifiability*, *Testability*
Portability	*Adaptability*, *Installability*

4.1 ISO: Performance Efficiency

The characteristics in the category *Performance efficiency* describe the performance of a system: how the system utilizes resources, responds to requests, and meets the capacity requirements. For two of the characteristics evidence was found.

Time Behavior - The first characteristic for which we found evidence is the time behavior of the system. For MDEEs, this is a special characteristic, because there are two main use cases for which the response and processing time is important. The run-time time behavior describes the response time of the functionality offered in the application. However, the second important use case for which response time is important, is the translation from model to application. When a generative approach is used, the model execution approach takes up time between model changes and software updates. When an interpretive approach is used, there is no time between model changes and software updates, because the execution happens during execution of normal system functions. These two distinct use cases are confirmed by the literature that we studied: we found comments in relation to both approaches. Therefore this characteristic is split into two separate characteristics. Both build-time time behavior and run-time time behavior are used as two separate characteristics in our study.

Twenty-two out of the 32 papers mention the time behavior characteristic, it is one of the most frequently commented characteristics. Because of the possibility of doing upfront analysis during code generation, more efficient code can be generated. On the other hand, interpreters add overhead to run-time functionality and thus are slower. While that is the general sentiment, Klint [34] remarked

Table 3. The results of the literature review and basis for the ranking of the two approaches. *G* corresponds with a preference for code generation over interpretation. *I* identifies where a paper shows a preference for interpretation over generation. *G I* indicates papers did not present a preference, but did give advantages or disadvantages.

	ISO: Performance behavior			**ISO: Compatibility**		**ISO: Security**	**ISO: Maintainability**				**ISO: Portability**	
	Run-time time behavior	Build-time time behavior	Resource utilization	Co-existence	Interoperability	Confidentiality	Modularity	Analysability	Modifiability	Testability	Adaptability	Installability
Batouta et al. [7]	G	I	G			G			I		G	G
Brady and Hammond [22]	G						I	I	I	I		
Cleenewerck [23]							G		G I			
Consel and Marlet [24]	G	I					I		I			
Cook et al. [25]	G		G						I			G
Cordy [26]		I							I			
Czarnecki and Eisenecker [4]	G	I										
Daz et al. [2]	G	I							I	G		
Ertl and Gregg [27]	G I	I							I		I	
Fabry et al.[8]	G		G		I				I			
Gaouar et al. [28]	G			I								
Gregg and Ertl [29]	G		G I					I	I	I	I	
Guana and Stroulia [10]								I	I	I		
Hinkel et al. [30]										I		
Inostroza and Van Der Storm [31]							I					
Jones et al. [32]	G							I	I	G		
Jrges [33]		I		I				I				
Klint [34]	G I		G					I	I	G I	I	
Meijler et al. [6]	G	I	G							G	G	G
Mernik et al. [17]							I		I	G		I
Ousterhout [35]	G	I			I					G		
Pessoa et al. [36]										G	I	
Riehle et al. [37]		I										
Romer et al. [38]	G I											
Schramm et al. [39]		I										
Stahl et al. [21]		I				G		I	I	I		
Sundharam et al. [40]								I		I	I	I
Tankovi [18]	G	I				G					I	I
Tankovi et al. [41]	G	I				G					I	I
Thibault et al. [42]	G								I			
Thibault and Consel [43]		I							I			
Varr et al. [44]	G				I					I		
Voelter [19]	G	I						G	G	G	G	G
Voelter and Visser [45]	G	I						G	G	G		
Zhu[46]	G	I	G							G		
% in favor of generation	88	0	87.5	0	0	100	20	20	15	55.5	30	50
% in favor of interpretation	12	100	12.5	100	100	0	80	80	85	44.5	70	50

that the overhead of interpreters will diminish with the advent in hardware. Both Ertl and Gregg [27] and Romer et al. [38] show that there is nothing that makes interpreters inherently slow.

The reduced build times that an interpretive approach gives are an advantage, such as enabling of agile development and better prototyping. This advantage is stated by Consel and Marlet [24] and Riehle et al. [37] among many others.

Resource Utilization - The general comment that code generation results in improved run-time behavior can be extended to resource utilization as well. Meijler et al. [6] state that generators can optimize for more than run-time behavior only, something that is useful in for instance embedded systems and game environments. A difference can also be seen in how generators or interpreters compete with the running application for resources. A generator might use more memory, but could be running on different hardware than the application. Interpreters are part of the application, so it could be hard to run them on different hardware. Gregg and Ertl [29] comment that interpreters often require less memory, but confirms the competition for resources with the application.

Another view on resource utilization is the data storage for an application. Meijler et al. [6] point out that the interpretive approach often leads to a less optimal data schema. The schema might depend on the model and thus can change at run-time, therefore, the schema has to be flexible enough. This requirement often conflicts with optimizations that might be achievable otherwise.

4.2 ISO: Compatibility

The category *Compatibility* contains characteristics that express the quality of co-existence and operability of the system.

Co-existence - Only two papers contain evidence for a preference between interpretation or generation based on this characteristic. Gaouar et al. [28] share their experiences on making dynamic user interfaces and point out how the interpretive approach enabled them to use platform native elements. A different side is shown in Jörges [33]: the late binding that interpretation offers makes it possible to re-use the same application instance for different tenants.

Interoperability - Interpreters have access to the dynamic context of the application at run-time. Fabry et al. [8], Ousterhout [35], and Varró et al. [44] state this as a preference for interpreters, because it allows them to communicate with the application in a way that is not possible by generators.

4.3 ISO: Security

Security describes the quality in terms of integrity, authentication, and confidentiality. The literature only contained evidence for the characteristic confidentiality.

Confidentiality - Tanković [18] and Tanković et al. [41] describe the models used in a MDEE as intellectual property. The interpretive approach exposes the model to the application, making it more vulnerable for exposure. In the generative approach the models do not need to be shipped which makes that approach more secure.

4.4 ISO: Maintainability

Maintainability is an important aspect in the quality of software products. Characteristics in this category that were mentioned by literature comment on the testability, modifiability, analysability, and modularity of the platform.

Modularity - Most literature favors interpreters over generation when looking at the modularity characteristic. Inostroza and Van Der Storm [31] and Consel and Marlet [24] propose solutions for modularization within interpreters. Cleenewerck [23] is the only one who argues that generators are more preferred than interpreters when it comes to modularization.

Analysability - An important aspect in MDEEs is the analysis of the resulting application. It should conform to the model and the defined semantics, which is not an easy task. When a generative approach is used, the model is translated in a separate language, without losing the semantics of the model. Proving that translation to be correct is hard, according to Guana and Stroulia [10]. According to Jörges [33], the interpreter can play the role of a reference implementation, used to document the semantics of the model. This improves the analysability of the platform.

Debugging is partly analyzing the run-time behavior of an application. According to Voelter [19] and Voelter and Visser [45] this process is easier in a generative approach, because the generated application can be debugged as if it were a normal application.

Modifiability - Many papers, Cook et al. [25] and Díaz et al. [2] among others, claim that interpreters are easier to write. We conclude that easier to write software is also easier to modify. Cordy [26] describes the process of a compiler as being heavy-weight, making it harder to modify. Cleenewerck [23] and Voelter and Visser [45] argue that generators give more freedom to developers, giving them room for better solutions.

Testability - The literature was far from conclusive on the testability of both approaches. On the one hand, interpreters can be embedded in test frameworks, this makes them easier to test. Generators on the other hand add indirection in the testing, because they are a function from model to code. Asserting the correctness of the output becomes fragile when just looking at the written code, the easiest way is to determine the correctness by running the code. Voelter [19] and Voelter and Visser [45] prefer generation when it comes to debugging, because the model translation can be left out of the testing.

4.5 ISO: Portability

Portability covers the characteristics adaptability and installability.

Adaptability - The separation between generation environment and application environment makes the generative approach preferred according to Meijler et al. [6], Batouta et al. [7], and Voelter [19]. The two environments can be evolved at a different pace when adaption needed, which makes it more flexible. In an interpretive approach the whole interpreter needs to be rewritten and although this might be easy, it is more work. However, Tanković [18], Tanković et al. [41], and Gregg and Ertl [29] state that porting an interpreter to a new platform is no problem when platform independent technologies (such as programming languages and environments that run on multiple platforms) are used. This matches the results from Sect. 3, where three SPOs stated portability as the rationale for the interpretive approach.

Installability - The two separated environments in the generative approach not only have a clear advantage for adaptability, it is also an advantage with respect to installability. Meijler et al. [6], Cook et al. [25], Batouta et al. [7], and Voelter [19] prefer code generation because it can target any platform, it does not constrain the target application. The initial installation is, however, not all that is important, when the MDEE is updated, re-installations are needed too. The interpretive approach makes re-installations less frequent, because in many cases only the model needs to be updated. This advantage is pointed out by Tanković [18] and Mernik et al. [17].

4.6 Utilizing the Preferences

The results of the literature study as presented in Table 3 can be used by SPOs to design their execution approach. But before SPOs can use these results, they have to prioritize the quality characteristics, i.e., they have to determine which characteristics are most important for them. When priorities are assigned, the preference for either the generative or the interpretive approach can be calculated by the following formulas:

$$P_{generative} = \sum_{i=1}^{12} P_i G_i \quad and \quad P_{interpretive} = \sum_{i=1}^{12} P_i I_i$$

The formulas summarize over all twelve characteristics i, and applies the priority (P_i) on the corresponding preference (from Table 3) for both the generative (G_i) and the interpretive (I_i) approach. All priorities add up to a total of 1, and because for every characteristic i G_i I_i add up to 100%, $P_{generative}$ and $P_{interpretive}$ add up to 100%. The outcome shows for a certain set of priorities what the preference for either the generative or interpretive approach is.

How the priorities are determined is not prescribed, however, in the case study described in the next Section we will show two possibilities. The first option is by informally giving a weight to every characteristic, dividing 100% among the different characteristics. By doing this informally, the SPO takes the risk of calculating a preference with inaccurate data. Therefore, we also show a second option to prioritize the characteristics: the analytic hierarchy process

(AHP) method described by Saaty [47]. Falessi et al. [48] shows that the AHP method is helpful in protecting against two difficulties that are relevant for this study. The first is a too coarse grained indication of the solution. When the priorities are determined informally it becomes easy to overlook certain characteristics. The second difficulty is that there are many quality attributes that need to be prioritized, and many attributes have small and subtle differences. The AHP method helps by prioritizing in a pairwise manner, the priorities are only determined relative to other characteristics.

5 Case Study

We conducted a case study by observing the design of a MDEE at a Dutch SPO, AFAS Software. The NEXT version of AFAS' ERP software is completely model-driven, cloud-based and tailored for a particular enterprise, based on an ontological model of that enterprise. The ontological enterprise model (OEM, see Schunselaar et al. [49]) will be expressive enough to fully describe the real-world enterprise of virtually any business. The platform initially used a generative approach, generating many lines of C# and JavaScript. However, during the course of 2016 a shift was put into motion towards a hybrid form with more parts being interpreted at run-time. We took part in the discussions surrounding this shift and observed the team while they designed and implemented parts of the MDEE.

We already explained that the context of the MDEE influences the design of the execution approach. This can be seen if we approach the architecture as a set of design decisions as described by Jansen and Bosch [12] and van der Ven et al. [13]. These decisions are made during the software development life cycle. Every requirement is satisfied by first creating one or more solutions, from which the SPO selects the best fitting alternative. This is done by assessing the solutions, for instance in terms of quality, cost, and feasibility. After a solution is selected, the preferred solution is incorporated in the existing architecture. This process is continuous and will be repeated for every new requirement that needs to be satisfied.

The complete architecture of a MDEE is too large to present in this paper, therefore, we present the most important and guiding requirements and decisions. These are presented in two distinct phases, to illustrate two different utilizations of the results from Table 3. The requirements and decisions that form the architecture and are input for the prioritization are summarized in Table 4.

The initial requirement that guided the design of the MDEE is the envisioned target audience for the modeling language (**R1**). By choosing laymen as the target audience, it becomes possible for non-technical business users to model their own ERP solution. This requirement is driven by years of experience in the development of an ERP solution, and the knowledge that is accumulated in those years. The resulting design decision is that the modeling language should be a model with a high level of abstraction, an ontological enterprise model (OEM) (**D1**). This model abstracts from the many details that are needed for

Table 4. Summary of the requirements and decisions from the design of the MDEE.

Requirements
R1 Target audience for the modeling language are laymen
R2 Users do not manage or maintain the MDEE themselves
R3 Cost effectiveness of the MDEE is important
R4 Use a technology that the developers are familiar with
R5 The MDEE should handle the load from the existing customer base
R6 End users can change the model without intervention
Decisions
D1 Develop an ontological enterprise model
D2 Use a SaaS delivery model
D3 Use multi-tenancy to gain resource sharing
D4 The MDEE should run on the .NET runtime
D5 Deploy the MDEE as a distributed application
D6 Use a hybrid execution approach

creating software, those details are added by the platform (the generator or interpreter) when the model is transformed. A second requirement is that the hosting and management of the MDEE is done by the SPO (**R2**). Delivering the MDEE through a Software-as-a-Service (SaaS) model is the second design decision (**D2**) that satisfies requirement **R2**. A third important requirement is cost effectiveness of the MDEE (**R3**), and multi-tenancy is one of the ways of achieving that as stated by Kabbedijk et al. [50]. The decision for a variant of multi-tenancy forms the last important decision (**D3**) of this initial phase.

After the design of the initial architecture, that solved among many other requirements **R1**, **R2**, and **R3**, the execution approach is designed. At the time of this design, the literature study as presented in Sect. 4 was not yet done. After discussion with the team, we concluded and verified that in hindsight four quality characteristics were especially important for this phase of the development. *Run-time time behavior* and *resource utilization* followed from the decision for SaaS (**D2**) and multi-tenancy (**D3**). *Testability* and *analysability* were important for AFAS to secure the quality of the new MDEE. With the data from Table 3 and the priorities that we assigned in hindsight allow us to calculate the preference for an approach. The possible calculation is shown as an illustration. The first two characteristics (*resource utilization* and *run-time time behavior*) are assigned a priority (or weight) of 35%, the other 30% is split between the other two characteristics (*testability* and *analysability*). The resulting preferences can then be calculated by combining the priorities of the characteristics with their weights (expressed in percentages, summing up to a total of 100%). We apply formulas $P_{generative}$ and $P_{interpretive}$ on the percentages from Table 3 and the priorities, resulting in the following calculations:

$$P_{generative} = 0.35 * 0.88 + 0.35 * 0.875 + 0.15 * 0.55.5 + 0.15 * 0.20 = 0.729$$

$$P_{interpretive} = 0.35 * 0.12 + 0.35 * 0.125 + 0.15 * 0.44.5 + 0.15 * 0.80 = 0.271$$

The outcome of the calculation matches the decision that AFAS made: their initial execution approach was the generative approach. This initial phase of requirements, decision making, and design of the architecture can be summarized in three statements.

- **R1** leads to **D1**
- **R2** in the context of **D1** leads to **D2**
- **R3** in the context of **D1** and **D2** leads to **D3**

As the design of the MDEE advanced new requirements needed to be realized. First of all the technology that is used to develop the MDEE was selected. The requirement was that a technology should be used that is familiar to the development team (**R4**). This fourth requirement led to the decision for the .NET runtime (**D4**) as the technology to develop the platform on. The next requirement formulated expected load requirements: AFAS has a large existing customer base that needs to be transferred to this new platform. There is an expected load known from the existing customer base that needs to be handled (**R5**). As a result of this requirement, the decision was made to design and deploy the application as a distributed system (**D5**).

Table 5. Summary of the priorities of quality characteristics determined by applying AHP as described by Saaty [47]. Columns *Generative* and *Interpretive* show the preferences for code generation and model interpretation from Table 3. The final preferences are calculated with the formulas $P_{generative}$ and $P_{interpretive}$.

	Priority	Generative	Interpretive
Run-time time behavior	0.059	0.88	0.12
Build-time time behavior	0.278	0.00	1.00
Resource utilization	0.098	0.875	0.125
Co-existence	0.045	0.00	1.00
Interoperability	0.012	0.00	1.00
Confidentiality	0.012	1.00	0.00
Modularity	0.062	0.20	0.80
Analysability	0.023	0.20	0.80
Modifiability	0.150	0.15	0.85
Testability	0.085	0.555	0.445
Adaptability	0.155	0.30	0.70
Installability	0.021	0.50	0.50
Preference		0.293	0.707

The sixth requirement reopened the design of the model execution approach. Therefore, the team decided to backtrack on the earlier decision for the generative approach. AFAS envisioned that customers are able to customize the model without intervention from AFAS (**R6**). This requirement leads to other requirements, such as the expected turn around time between model changes and application updates. Based on requirement **R6** and the decisions **D1-D5** the quality characteristics were prioritized. Characteristics *build-time time behavior*, *adaptability*, and *modifiability* became more important. This time the prioritization was done by applying the AHP method: all the characteristics were pair-wise compared and ranked according to the method described by Saaty [47]. The results are shown in Table 5, combined with the preferences from Table 3. The final outcome preferred interpretation over generation with 71%.

The team decided to implement a simplification approach: the OEM is simplified into a simpler model by the generator. This way the team was able to satisfy the build-time time requirements, without sacrificing performance. Because the MDEE itself already grew quite large, the team decided to also switch to a mix-and-match approach. The simplification approach was first implemented in a specific component: the messages that are past between the different parts of the distributed system.

An architecture consists of many decisions, both large and small, both important and non-essential. Our case study only shows the five most important requirements. In the next Section we will reflect on the case study and derive a proposed decision support framework for the design of a model execution approach.

6 Case Study Reflection

In Sect. 5 we observed a SPO during the design of a MDEE. We have shown how design decisions from the architecture determine the priorities of the quality characteristics. The existing architecture of the MDEE and the design decisions that are present together form the context of the model execution approach. It shows that, just as with any component in a larger system, the design of an execution approach does not stand on its own, but needs to be embedded in the overall architecture. Some design decisions might constrain the execution approach, other design decisions might even mitigate the problems that an execution approach give. As an example we look at build-time time behavior, a requirement that was described in the previous Section. From Table 3 we learn that the interpretive approach is preferred when a specific build-time time behavior is required. However, when the MDEE will be built using a programming language and platform that uses interpretation, such as JavaScript, the decrease in build times with a generative approach might be mitigated. An interpreted language does not need a separate compile step that needs to be executed by the generator, and that reduces the build time. This shows that the design decisions that are already present influence the execution approach.

We have distilled three areas from the decisions described in Sect. 5 that steered the design of the model execution approach. The decisions described in Sect. 5, and summarized in Table 4 are used to illustrate the areas.

6.1 The Metamodel

The metamodel and its features and requirements have an influence on the most fitting model execution approach. This is illustrated by decision **D1: OEM** and requirement **R6: Customize the model**.

A model with a high-level of abstraction (such as **D1: OEM**) will require a more complex model execution, because the distance in terms of abstraction between a programming language and the model is larger. With an interpretive approach, the application will require more resources to perform this model execution. This influences the run-time behavior of the model execution approach, and thus the application itself.

On the other hand, requirement **R6: Customize the model** increases the priority of the build-time time behavior characteristic. This leads to a preference for run-time interpretation, because that approach is preferred if build-time time behavior is important.

6.2 The Architecture

The chosen architecture for the application forms a second area of influence on the most fitting model execution approach. A multi-tenant, distributed application (as defined by **D3: Multi-tenancy** and **D5: Distributed application**) can result in conflicting requirements for the most fitting model execution approach.

On the one hand, multi-tenancy prefers interpretation, because it allows the sharing of a single application instance for multiple tenants (see characteristic co-existence in Sect. 4). This maximizes the resource sharing, and enables fast unloading and loading of changes, which decreases the build times. On the other hand, a distributed application might not benefit from interpretation, because every process has to do the interpretation. Figure 2 shows that the interpretation process is part of the application, and is thus duplicated when the application is separated in multiple components and processes. This adds of course resource utilization to the platform.

The decision for a distributed application (**D5: Distributed application**), makes it possible to design a hybrid model execution approach. A distributed application consists of different (distributed) components that can use their own execution approach, shown in Sect. 5 where only the messages were re-designed.

6.3 The Platform

Although Kelly and Tolvanen [5] make no distinction between the architecture, framework, the operating system, or the runtime environment, we see a different influence from the operating system or runtime environment. As decision **D4: .NET platform** illustrates, the lack of support for dynamic software updating requires a different model execution approach to satisfy the requested build-time time behavior. This matches the approaches of Meijler et al. [6] with their customized Java class loader and Czarnecki and Eisenecker [4] using the extension object pattern.

The SaaS delivery model (**D2: SaaS delivery model**) removes most of the problems around *installability* and *co-existence*: the platform is controlled by the SPO.

6.4 The Decision Support Framework

From the observations we see three distinct areas that influence the model execution approach. The metamodel and its features and requirements lead to decisions that influence the execution approach. The architecture and the platform can both constrain the execution approach as well as mitigate challenges. Determining the priorities for the quality characteristics can be a difficult task.

The design of the best fitting model execution approach for a MDEE is not different from other parts of the MDEE; it is not possible without knowledge of the context. The description of the design process that we gave in Sect. 5 is generic for the software development life cycle. We propose, based on the observations made during the case study, a tailored version of the process for the design of a model execution approach (shown in Fig. 3). It shows that the current architecture is input for the prioritization of the quality characteristics. The priorities can then be used to asses the possible execution approaches. How the priorities are determined is not prescribed by the framework, however, we have shown two possible methods to determine them: an informal method and the AHP method.

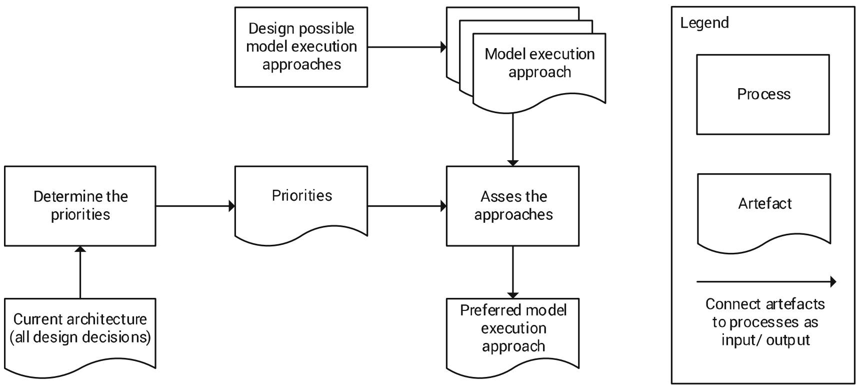
Design possible model execution approaches
Model execution approach
Legend
Process
Determine the priorities
Priorities
Asses the approaches
Artefact
Current architecture (all design decisions)
Preferred model execution approach
Connect artefacts to processes as input/ output

Fig. 3. The process of selecting a best fitting model execution approach. The process starts with the design of possible execution approaches. The current architecture is the input for the prioritization of quality characteristics. The priorities can be used in assessing possible execution approaches.

The framework offers guidance for SPOs in the design of their model execution approach. By formalizing their architecture in a set of design decisions, and by prioritizing the quality characteristics, SPOs can calculate the preference for either the generative or the interpretive approach. This can then in turn be used to design a fitting hybrid model execution approach.

7 Discussion

The validity of our research is threatened by several factors. The internal validity of our study is threatened because the correlation between quality characteristics on te one hand and the execution approach on the other hand are not straightforward. The claims in the reviewed literature, however, do show a convergence towards each other. Some characteristics lack a significant number of references, making them volatile. However, we regard the claims that are made not as controversial, but in line with existing research. The data that we found in literature consists of anecdotal argumentation, based on the experience of the authors. The claims that were made, were not validated and not supported with empirical evidence. To create a more trustworthy decision support framework, the data presented in Table 3 should be validated by empirical research. Experiments or large case studies should provide more quantitative data on the fulfillment of the different quality characteristics.

The construct validity of our case study is threatened by the fact that one of the authors is involved in the object of the study, resulting in a possible bias in our observations. However, the observations were made during a period of several months in which the model execution was actively designed. Our observations were reviewed and commented on by other team members involved. The descriptions of the observations, and the described requirements and decisions were correctly described according to these comments.

The external validity of our research is threatened because our case study is done at a single company. The observations, however, were done over an extensive period of time, and the results were discussed with the team. We argue that the conclusions and observations are in line with existing literature. The decision support framework, however, should be further strengthened by additional case studies.

8 Conclusion

We present two contributions to the research on MDD, and in particular the development of MDEEs. The survey in Sect. 3 illustrates that there is a lack of guidance and knowledge for SPOs. Although the SPOs show that indeed many forms of model execution approaches are used, they do not have an explicit rationale for their design.

In Sect. 4 we studied and summarized existing literature to correlate quality characteristics with model execution approach. Although this knowledge was already available, it was scattered over many papers. Our study makes the experience and knowledge of many authors available to MDD researchers and practitioners. We summarized the results in Table 3, which can be used as a reference in the design of a fitting model execution approach. In Sect. 5 we demonstrate how these results can be used as input for the decision making in selecting alternatives.

The second contribution that we present is the decision support framework as presented in Sect. 6. With this framework, SPOs have a structured process for the design of the model execution approach By making these design decisions explicit, and by adding the results from Table 3 as input to the decision making process, SPOs can design the best fitting execution approach. The influence of the context of the MDEE as shown in Sect. 6, and the interplay between existing design decisions and the model execution approaches is made explicit and can lead to better designs.

Although we are not able to relieve SPOs from the hard work of designing a model-driven engineering environment, we argue that our research brings them closer to the best fitting design. By making existing knowledge and experience accessible, the solutions in the decision making process can be assessed with more confidence. In Sect. 3 we show that many SPOs already use a hybrid form of model execution, but do not have a strong rationale. However, our research also uncovers the need for more empirical research to support SPOs in the design and development of MDEEs. Table 3 is primarily based on anecdotes, and often not backed by real evidence. Experiments and case studies should be conducted to strengthen the evidence used in our decision support framework. The framework itself is created by observing a single SPO designing a model execution approach, and it should be evaluated by applying it at other SPOs.

Many questions in the design of software can be answered with "it depends", leaving the questioner puzzled as to what he should do. We present how the context of the MDEE influences the design of a model execution approach for MDEEs. Existing design decisions determine the priorities of quality characteristics, which in term steer the design of the model execution approach. We also show how SPOs can utilize the knowledge presented in this paper to allow them to steer their design process towards the most fitting model execution approach.

Acknowledgements. This research was supported by the NWO AMUSE project (628.006.001): a collaboration between Vrije Universiteit Amsterdam, Utrecht University, and AFAS Software in the Netherlands. The NEXT Platform is developed and maintained by AFAS Software. Further more, the authors like to thank Jurgen Vinju, Tijs van der Storm, and their colleagues for their feedback and knowledge early on in the writing process. Finally we thank the team at AFAS Software for their opinions, feedback, and reviews.

References

1. Overeem, M., Jansen, S.: An exploration of the 'It' in 'It Depends': generative versus interpretive model-driven development. In: 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD (2017)

2. Díaz, V.G., Valdez, E.R.N., Espada, J.P., Bustelo, B.C.P.G., Lovelle, J.M.C., Marín, C.E.M.: A brief introduction to model-driven engineering. Tecnura **18**, 127–142 (2014)
3. Brown, A.W.: An Introduction to Model Driven Architecture. The Rational Edge, pp. 1–16 (2004)
4. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, Boston (2000)
5. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, Hoboken (2008)
6. Meijler, T.D., Nytun, J.P., Prinz, A., Wortmann, H.: Supporting fine-grained generative model-driven evolution. Softw. Syst. Model. **9**(3), 403–424 (2010)
7. Batouta, Z.I., Dehbi, R., Talea, M., Hajoui, O.: Multi-criteria analysis and advanced comparative study between automatic generation approaches in software engineering. J. Theor. Appl. Inf. Technol. **81**, 609–620 (2015)
8. Fabry, J., Dinkelaker, T., Noye, J., Tanter, E.: A taxonomy of domain-specific aspect languages. ACM Comput. Surv. **47**, 1–44 (2015)
9. Zhu, L., Aurum, A., Gorton, I., Jeffery, R.: Tradeoff and sensitivity analysis in software architecture evaluation using analytic hierarchy process. Softw. Qual. J. **13**(4), 357–375 (2005)
10. Guana, V., Stroulia, E.: How do developers solve software-engineering tasks on model-based code generators? An empirical study design. In: First International Workshop on Human Factors in Modeling (2015)
11. Capilla, R., Rey, U., Carlos, J., Dueñas, J.C., Madrid, U.P.D.: The decision view's role in software architecture practice. IEEE Softw. **26**(2), 36–43 (2009)
12. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), pp. 109–120 (2005)
13. van der Ven, J.S., Jansen, A.G.J., Nijhuis, J.A.G., Bosch, J.: Design decisions: the bridge between rationale and architecture. In: Dutoit, A.H., McCall, R., Mistrík, I., Paech, B. (eds.) Rationale Management in Software Engineering, pp. 329–348. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-30998-7_16
14. Svahnberg, M., Wohlin, C., Lundberg, L., Mattsson, M.: A quality-driven decision-support method for identifying software architecture candidates. Int. J. Softw. Eng. Knowl. Eng. **13**, 547–573 (2003)
15. Wohlin, C.: Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: 18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014), pp. 1–10 (2014)
16. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Not. **35**, 26–36 (2000)
17. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**, 316–344 (2005)
18. Tanković, N.: Model driven development approaches: comparison and opportunities. Technical report (2011)
19. Voelter, M.: Best practices for DSLs and model-driven software development. J. Object Technol. **8**, 79–102 (2009)
20. ISO: ISO/IEC 25010:2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Standard, International Organization for Standardization, Geneva, CH (2011)
21. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management (2006)

22. Brady, E.C., Hammond, K.: Scrapping your inefficient engine. ACM SIGPLAN Not. **45**, 297 (2010)
23. Cleenewerck, T.: Modularizing language constructs: a reflective approach. Ph.D. thesis (2007)
24. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. Princ. Declar. Program. **1490**, 170–194 (1998)
25. Cook, W.R., Delaware, B., Finsterbusch, T., Ibrahim, A., Wiedermann, B.: Model transformation by partial evaluation of model interpreters. Technical report (2008)
26. Cordy, J.R.: TXL - a language for programming language tools and applications. In: Proceedings of the ACM 4th International Workshop on Language Descriptions, Tools and Applications, pp. 1–27 (2004)
27. Ertl, M.A., Gregg, D.: The structure and performance of efficient interpreters. J. Instr.-Level Parallelism **5**, 1–25 (2003)
28. Gaouar, L., Benamar, A., Bendimerad, F.T.: Model driven approaches to cross platform mobile development. In: Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication, pp. 19:1–19:15 (2015)
29. Gregg, D., Ertl, M.A.: A language and tool for generating efficient virtual machine interpreters. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 196–215. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25935-0_12
30. Hinkel, G., Denninger, O., Krach, S., Groenda, H.: Experiences with model-driven engineering in neurorobotics. In: Wąsowski, A., Lönn, H. (eds.) ECMFA 2016. LNCS, vol. 9764, pp. 217–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_14
31. Inostroza, P., Van Der Storm, T.: Modular interpreters for the masses implicit context propagation using object algebras. ACM SIGPLAN Not. **51**(3), 171–180 (2015)
32. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall International (1993)
33. Jörges, S.: Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach, vol. 7747. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36127-2
34. Klint, P.: Interpretation techniques. Softw.: Pract. Exp. **11**, 963–973 (1981)
35. Ousterhout, J.K.: Scripting: higher-level programming for the 21st century. Computer **31**, 23–30 (1998)
36. Pessoa, L., Fernandes, P., Castro, T., Alves, V., Rodrigues, G.N., Carvalho, H.: Building reliable and maintainable dynamic software product lines: an investigation in the body sensor network domain. Inf. Softw. Technol. **86**, 54–70 (2017)
37. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: International Conference on Object Oriented Programming Systems Languages and Applications (OOSPLA), pp. 327–341 (2001)
38. Romer, T.H., Lee, D., Voelker, G.M., Wolman, A., Wong, W.A., Baer, J.L., Bershad, B.N., Levy, H.M.: The structure and performance of interpreters. ACM SIGPLAN Not. **31**, 150–159 (1996)
39. Schramm, A., Preußner, A., Heinrich, M., Vogel, L.: Rapid UI development for enterprise applications: combining manual and model-driven techniques. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 271–285. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_19

40. Sundharam, S.M., Altmeyer, S., Navet, N.: Model interpretation for an AUTOSAR compliant engine control function. In: 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) (2016)
41. Tanković, N., Vukotić, D., Žagar, M.: Rethinking model driven development: analysis and opportunities. In: Proceedings of the ITI 2012 34th International Conference on Information Technology Interfaces (ITI), pp. 505–510 (2012)
42. 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**, 363–377 (1999)
43. Thibault, S., Consel, C.: A framework for application generator design. ACM SIGSOFT Softw. Eng. Notes **22**, 131–135 (1997)
44. Varró, G., Anjorin, A., Schürr, A.: Unification of compiled and interpreter-based pattern matching techniques. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 368–383. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_28
45. Voelter, M., Visser, E.: Product line engineering using domain-specific languages. In: 15th International Software Product Line Conference, pp. 70–79 (2011)
46. Zhu, M.: Model-driven game development addressing architectural diversity and game engine-integration. Ph.D. thesis (2014)
47. Saaty, T.: How to make a decision: the analytic hierarchy process. Eur. J. Oper. Res. **48**, 9–26 (1990)
48. Falessi, D., Cantone, G., Kazman, R., Kruchten, P.: Decision-making techniques for software architecture design. ACM Comput. Surv. **43**, 1–28 (2011)
49. Schunselaar, D.M.M., Gulden, J., Schuur, H.V.D., Reijers, H.A.: A systematic evaluation of enterprise modelling approaches on their applicability to automatically generate software. In: 18th IEEE Conference on Business Informatics, pp. 290–299 (2016)
50. Kabbedijk, J., Bezemer, C.P., Jansen, S., Zaidman, A.: Defining multi-tenancy: a systematic mapping study on the academic and the industrial perspective. J. Syst. Softw. **100**, 139–148 (2015)