An Exploration of the ‘It’ in ‘It Depends’: Generative versus Interpretive Model-Driven Development

Michiel Overeem[1] and Slinger Jansen[2]

[1]*Department Architecture and Innovation, AFAS Software, Leusden, The Netherlands*

[2]*Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands*

m.overeem@afas.nl, slinger.jansen@uu.nl

Keywords: Model-Driven Development, Model-Driven Architecture, Software Architecture, Code Generation, Run-Time Model Interpretation, Decision Support Making

Abstract: Software producing organizations are increasingly using model driven development platforms to improve software quality and developer productivity. Software architects, however, need to decide whether the platform generates code (that might be compiled) or if the model is immediately interpreted by an interpreter embedded in the application. Presently, there is no clear guidance that enables architects to decide for code generation, interpretation, or a hybrid approach. Although the approaches are functionally equivalent, they have different quality characteristics. An exploration is done on the quality characteristics of code generation versus interpretation as a model execution approach. A literature study is done to gather quantitative data on the quality characteristics of the two model execution approaches. The results of this study are matched with observations made during a case study. With the resulting support method architects of model driven development platforms can avoid costly wrong choices in the development of a model driven development platform.

1 INTRODUCTION

Software producing organizations (SPOs) are using model driven development (MDD) platforms to improve software quality and developer productivity as described by Díaz et al. (2014). The promise of MDD is an increase of velocity for a development team, tooling for non-technical employees to specify functionality and intent, and provable correctness of an application. According to Hailpern and Tarr (2006) this is achieved by raising the abstraction level at which developers work. However, implementing MDD in a SPO is not trivial. A model that is transformed into running software sounds as a major step forward, but how and when is this transformation done? Some implementations use code generation, others use run-time interpretation, but which one is better? In line with Voelter (2009), who calls this transformation the execution of the model, this paper uses the term *model execution approach*: the model is executed to obtain the software that conforms to the model.

Different authors such as Batouta et al. (2015), Smolik and Vitkovsky (2012), Tankovic (unknown), and Voelter (2009) have described possible model execution approaches and how they should be implemented. It is clear that trade-offs have to be made. No one should expect that the model execution approach for one project is suitable for another project, without exploring the trade-offs first. Developers have to find the balance, and the answer remains *“it depends”*.

One can be tempted to regard this as a mere implementation detail and misuse quotes like “any good software engineer will tell you that a compiler and an interpreter are interchangeable”[1] and “interpreters and generators are functionally equivalent” as stated by Stahl et al. (2006). However, that would miss the point of this discussion: the different model execution approaches will give the same functionality, but not with the same level of quality. Although Stahl et al. (2006) give a rule of thumb by saying that code generation is more useful for structural aspects, while interpretation is better suited for behavioral aspects, this paper shows that there is more to say about the design of a fitting model execution approach. The challenge for SPOs is to implement the model execution approach in such a manner that their required qual-

⅄ This is an AMUSE paper. See amuse-project.org for more information.

[1]Tim Berners-Lee in an interview with editor Brian Runciman - http://www.bcs.org/content/ConWebDoc/3337

ity attributes of the system are satisfied with the least amount of effort. Although the model execution approaches are functionally equivalent and an end-user will not see the difference the quality of the system depends on this part as well. Interpretation for instance can negatively affect the run-time performance as explained in Section 3. Code generation on the other hand gives ample opportunity for optimization of the resulting application.

This paper deals with the foundational question of whether to generate running software or interpret the model at run-time. The context of this research is a large scale MDD platform in development at AFAS Software, a SPO in The Netherlands. The model execution approach is part of this platform, and the design of this approach is the motive of this research. The decision-making process is explored by answering two questions: *How does the choice between generative and interpretive MDD influence the quality of a MDD platform?* and *How can SPOs design the most fitting model execution approach based on the quality characteristics for generative and interpretive MDD?* By answering these two questions, the trade-offs are made explicit, resulting in decision support for the design of a model execution approach.

Section 2 gives a brief overview of different model execution approaches, and summarizes related work. The first question is answered in Section 3 by performing a literature study on the advantages and disadvantages of the generative and interpretive approach, and relating them to the Software Product Quality Model from ISO/IEC 25010 (2011). Section 4 answers the second question by observing the decision-making process. The research is evaluated and discussed in Sections 5 and 6.

2 RELATED WORK

The first and maybe best-known approach is *code generation*, a strategy in which a model is parsed, interpreted, and transformed into running software by generating source code. This approach is formalized outside of MDD in *Generative Programming* and defined by Czarnecki and Eisenecker (2000): "Generative programming is a software engineering paradigm based on modeling software system facilities such that a highly customized and optimized intermediate or end-product can be automatically manufactured on demand." Generative programming is more broad than MDD: the generation does not need to happen based on an model, but can also be done based on a configuration of components. Combining generative programming with MDD results in generative MDD. Although the output could be manually changed, this paper only regards *full* code generation: no manual changes are made to the resulting source code between generation and shipping the application. As Kelly and Tolvanen (2008) point out this does not mean that all code should be generated, an approach that combines generated code with a framework or base library is still a full code generation approach.

The other well-known approach is *model interpretation* or *interpretive MDD*. This strategy also parses and interprets the model; however, in this approach this is done at run-time. There is no source code or application generated and deployed, but instead, the model is shipped as meta-data for the application. In the same manner as the generation approach, this paper only regards interpretation without further manual coding. Interpretation which needs manual coding would not make sense, as model execution happens at run-time there would be no time for intervention.

Although many unique hybrid approaches could be identified, three general groups of hybrid approaches are explained. Because of the nature of the hybrid approach, a mix of different approaches, it is not possible to give an exhaustive overview. The first hybrid approach is simplification: a model with a high abstraction level is transformed into a model with a lower level of abstraction. The generator simplifies the original model and the resulting model, of a lower abstraction level, is interpreted at run-time. This is very similar to languages that compile into an intermediate language, that in return is interpreted in a runtime environment. An example of this approach is described by Meijler et al. (2010). They generate Java code, but use a customized class loader to dynamically load classes. The customized class loaders acts as an interpreter while the generated Java code is the intermediate language. The approach of a customized class loader looks similar to the Adaptive Object-Model Architecture described by Yoder and Johnson (2002) and applied by Hen-Tov et al. (2008).

The second group of hybrid approaches is a strategy that combines different approaches to different parts of the model and/or application. In this approach code is generated for certain parts of the system, while other parts of the application are built using interpreters. The separation could also be done from the model perspective: generate code for the mature (and thus more stable) parts of the model while parts that are more dynamic can be interpreted.

The third hybrid approach is found in programming language research: while Jones et al. (1993) and Rohou et al. (2015) show that interpretation is slower, *partial evaluation* opens up new possibilities in implementing a model execution strategy. A com-

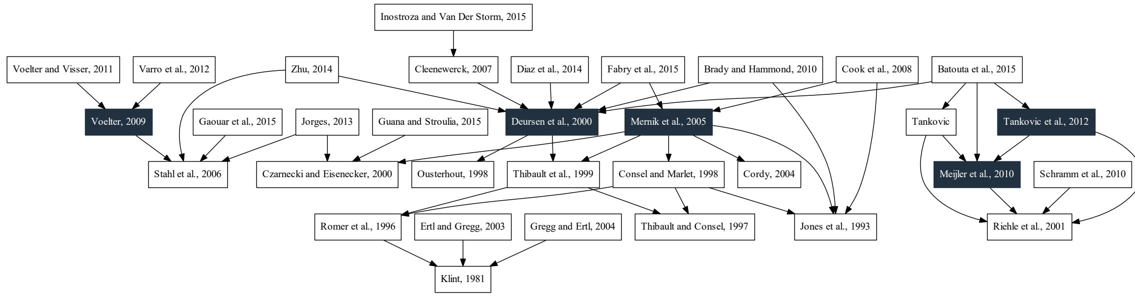

Figure 1: A dependency graph of the literature found in the review. Every arrow represents a citation. The dark blue boxes mark the start set for the literature study.

pany that started with an interpretation approach, is able to specialize this interpreter for certain models. The evaluator transforms the generic interpreter into a more specialized interpreter for a certain range of models. Cook et al. (2009) and Shali and Cook (2011) demonstrate that with this approach the interpreter could even be translated into a specialized application.

More research is done on model execution approaches in MDD, however, none of them answer the questions asked in this paper. Batouta et al. (2015) have performed a multi-criteria analysis on the different approaches to aid the decision-making. However, they do make a decisive statement about which approach is better without giving support for decision-making in different contexts. Fabry et al. (2015) touch on a number of advantages regarding the different model execution approaches, but do this without giving any decision-making support. While Zhu et al. (2005) does research the decision-making, he does this for other architectural decisions than the specific model execution approach. Guana and Stroulia (2015) research how developers interact with code generators, but do not compare this with interpreters.

Applying different decision-making methods to the design of software and their architecture is not a new approach. Examples are Capilla et al. (2009), Jansen and Bosch (2005), and Svahnberg et al. (2003). All of them research how decision-making methods can support the design of software. None of them apply this approach to the architecture of MDD in particular, but only look at architectural design in general.

3 LITERATURE STUDY

How the two model execution approaches, code generation and interpretation, compare to each other is studied by doing a literature review following the snowballing approach as described by Wohlin (2014). The goal of the study is to identify advantages and disadvantages of generation and interpretation that are claimed in existing research. There are two reasons for using the snowballing strategy. First of all the research area to be covered is broad: MDD, DSL engineering, and compiler design all cover aspects of this discussion. The second reason is that the papers often do not criticize the approaches explicitly, advantages or disadvantages are often buried in descriptions of implementations. These two reasons, the broad research area and the indirect comments, make it hard to search for literature. The snowballing strategy depends on the citations that authors add when giving reasons for their chosen approach. Therefore, the start set consists of papers that for instance propose a specific implementation or give a summary of best practices.

The start set for the review was created by an informal exploratory search using Google. Search terms for this search were “interpretation versus code generation” as well as a number of variations. The results were reviewed and explored for links to scientific research. The resulting literature from the different research fields was taken as starting point: Van Deursen et al. (2000), Meijler et al. (2010), Mernik et al. (2005), Tanković et al. (2012), and Voelter (2009). Both backward and forward references were followed to expand this set. Papers were included when advantages or disadvantages were mentioned in relation to a model execution approach. This strategy led to 32 papers and books that were included. The citation graph is shown in Figure 1.

The classification of the advantages and disadvantages is done by using the characteristics for software product quality found in ISO/IEC 25010 (2011). The quality attributes form the criteria of the multi-criteria decision support method that is

described in this paper. The product quality model as defined by ISO consists of eight categories (with 31 sub-characteristics).

In the 32 papers, no evidence was found for a difference in quality fulfillment in relation to three out of eight categories. It was expected that those three categories would not have any evidence of quality difference. Categories *functional suitability* and *usability* describe how users interact with the software and how the software fulfills their requirements, and to repeat the quote from Stahl et al. (2006): code generation and model interpretation are functionally equivalent. The last category, *reliability*, deals with behavior in relation to the environment of the software system.

For the remaining categories data was found. The category *performance efficiency* describes how to system utilizes resources, responds to requests, and meets the capacity requirements. The operability and co-existence of a system are described in *compatibility*. Confidentially, integrity, authentication, and related aspects are described in *security*. *Maintainability* covers the effectiveness and efficiency of modifications to the system. Last, *portability* describes the transfer of the system to a new platform. Mentions of an advantage or disadvantage were mapped onto sub-characteristics of these categories.

Table 1 shows the results of the study: every mention of an advantage or disadvantage results in either a *G* (when the author prefers generation over interpretation) or an *I* (when the author prefers the opposite) in the corresponding cell. When the author has no preference for generation over interpretation, but does give advantages for both, the corresponding cell contains *G I*. The number of papers that prefer generation over interpretation or vice versa are used to calculate the preference in a percentage of the two alternatives with respect to every quality attribute. The resulting ranks are used in Section 4 to design the most fitting approach.

The evidence found is summarized per sub-characteristic. First the name of the sub-characteristic, the category under which the sub-characteristic falls between parentheses, and a short summary of the evidence that was found.

Time behavior (Performance efficiency) - This sub-characteristic is a special one, as according to ISO it deals with the response and processing time of the system. Within a MDD platform there are two different response times that are important: the run-time response of the application and the build-time response of the platform. Build-time response is defined as being the turnaround time of changing the model and updating the application to correspond with the model. The literature shows that there is a big difference between those two kinds of response times when comparing code generation with model interpretation. With respect to the run-time performance of the application a big preference for generation is observed, while for build-time performance there is a big preference for interpretation. Because of this difference the time behavior sub-characteristic is split out in two separate characteristics.

The trade-off between interpretation and generation with respect to run-time performance is a frequently made comment. Twenty-two papers list this trade-off and make comments about it: three give no preference, and the others prefer a generative approach. The general sentiment is that generators can do upfront analysis resulting in more efficient code, while interpreters add overhead and thus are slower. In an early paper Klint (1981) went against this sentiment and remarked that this advantage will diminish over time because of the advent in hardware. Ertl and Gregg (2003) and Romer et al. (1996) indeed show in their research that there are no reasons why interpreters are slow by definition.

Although the generative approach often results in better run-time performance, the lower build times are an advantage of the interpretive approach. The turnaround time between model changes and ready-for-use software is a huge advantage, enabling better prototyping, according to Consel and Marlet (1998), and Riehle et al. (2001) among many others.

Resource utilization (Performance efficiency) - Not only are generators producing more performant code, they can also optimize for other resources according to Meijler et al. (2010). This is a much needed advantage in for instance embedded systems or other resource-constrained environments like games. Gregg and Ertl (2004) state that interpreters often require less memory, but they do compete with other parts of the application for resources. Although generators maybe use more memory, they do not have to compete with the running application, they can run on different hardware. Meijler et al. (2010) point out that with regards to data storage in the application an interpretive approach often has to choose for a less optimal schema. The data itself conforms to a schema depending on the model, which can change at run-time. The storage often can not respond to that change and thus has to be flexible enough.

Co-existence (Compatibility) - Only two papers make comments on the influence of the different approaches on co-existence. Jörges (2013) points

Table 1: 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* shows where papers did not show a clear preference, but did give advantages or disadvantages.

	Run-time behavior	Build-time behavior	Resource utilization	Co-existence	Interoperability	Confidentiality	Modularity	Analysability	Modifiability	Testability	Adaptability	Installability
Batouta et al. (2015)	G	I	G			G			I		G	G
Brady and Hammond (2010)	G						I	I	I	I		
Cleenewerck (2007)							G		G I			
Consel and Marlet (1998)	G	I					I		I			
Cook et al. (2008)	G		G						I			G
Cordy (2004)		I							I			
Czarnecki and Eisenecker (2000)	G	I										
Díaz et al. (2014)	G	I							I	G		
Ertl and Gregg (2003)	G I	I							I		I	
Fabry et al. (2015)	G		G		I				I			
Gaouar et al. (2015)	G			I								
Gregg and Ertl (2004)	G		G I					I	I	I	I	
Guana and Stroulia (2015)								I	I	I		
Inostroza and Van Der Storm (2015)							I					
Jones et al. (1993)	G							I	I	G		
Jörges (2013)		I		I				I				
Klint (1981)	G I		G					I	I	G I	I	
Meijler et al. (2010)	G	I	G							G	G	G
Mernik et al. (2005)							I		I	G		I
Ousterhout (1998)	G	I			I					G		
Riehle et al. (2001)		I										
Romer et al. (1996)	G I											
Schramm et al. (2010)		I										
Stahl et al. (2006)		I				G		I	I	I		
Tankovic (unknown)	G	I				G					I	I
Tanković et al. (2012)	G	I				G					I	I
Thibault et al. (1999)	G								I			
Thibault and Consel (1997)		I							I			
Varró et al. (2012)	G				I					I		
Voelter (2009)	G	I						G	G	G	G	G
Voelter and Visser (2011)	G	I						G	G	G		
Zhu (2014)	G	I	G							G		
% in favor generation	88	0	87.5	0	0	100	20	22.2	15	60	37.5	57.1
% in favor interpretation	12	100	12.5	100	100	0	80	77.8	85	40	62.5	42.9

out that the late binding of the interpretive approach makes it a good candidate for multi-tenant applications: the same application instance can be used for all tenants. Gaouar et al. (2015) share their experiences on making dynamic user interfaces and point out how the interpretive approach enabled them to use the platform native elements.

Interoperability (Compatibility) - Because interpreters have access to the dynamic context of the running application they can make decisions based on that context. This advantage is claimed by Fabry et al. (2015), Ousterhout (1998), and Varró et al. (2012). The interoperability of interpreters with other parts of the application is thus better.

Confidentiality (Security) - Models can be seen as intellectual property according to Tankovic (unknown) and Tanković et al. (2012), and the model is exposed to the application in an interpretive approach. With code generation the models are never shipped, as the modeling solution is separated from the application.

Modularity (Maintainability) - In his thesis Cleenewerck (2007) argues that generators have more room for modularization, because an interpreter at some point produces a value which cannot be changed anymore. He is however the only one claiming this preference, different authors such as Inostroza and Van Der Storm (2015) and Consel and Marlet (1998) propose solutions for modularization within interpreters.

Analysability (Maintainability) - Generators translate the model into a separate language, but the semantics have to be translated as well. This is hard to prove correct, according to Guana and Stroulia (2015). Interpreters on the other hand can play the role of a reference implementation (according to Jörges (2013)), making the semantics of a model clear. Voelter (2009) and Voelter and Visser (2011) claim an advantage for generation: debugging a generated application is easier, which benefits the analysability.

Modifiability (Maintainability) - Cook et al. (2008) and Díaz et al. (2014), among others, claim that interpreters are easier to write, and thus are easier to modify. In the experience of Cordy (2004)

the heavy-weight process of the compiler made it hard to modify. Cleenewerck (2007) and Voelter and Visser (2011) claim that developers have more freedom building generators and thus can make better maintainable solutions.

Testability (Maintainability) - Interpreters can easily be embedded in a test framework, where the functionality can be tested. The reduced turnaround time for interpreters also make them easier to test. Generators are effectively a function from model to code, which results in an indirection in the testing framework. The easiest way to determine the correctness is by running the generated code, asserting if the correct code is generated would be cumbersome and error prone. Voelter (2009) and Voelter and Visser (2011) give preference to generation when it comes to debugging, because only one aspect is being looked at.

Adaptability (Portability) - Meijler et al. (2010), Batouta et al. (2015), and Voelter (2009) see the advantage of code generation in adaptability: only part of the generator needs to be adapted, which is less work. Because of the separation between generator environment and application environment it is possible to evolve them in different steps. Tanković et al. (2012) and Gregg and Ertl (2004) see no problem in porting an interpreter to a new platform by using platform independent technologies. When one wants to move to a different target platform, worst case scenario is that the whole interpreter needs be rewritten, which some regard as being easy, while with the generative approach only part of the generator needs to be adapted, which can be less work.

Installability (Portability) - A clear advantage is given to code generation by Meijler et al. (2010), Cook et al. (2008), Batouta et al. (2015), and Voelter (2009), because the generation can target any platform. By using an interpretative approach it is less needed to re-install the application, because only the model needs to be updated. This advantage is pointed out by Tanković et al. (2012) and Mernik et al. (2005).

4 CASE STUDY

The quality characteristics and their comparisons can be used to design the most fitting approach for a specific software product or software component. In order to make usage of them, teams need to prioritize the characteristics, i.e., they have to determine which are more important than others. This section summarizes observations made at a large software company in The Netherlands. AFAS Software, a Dutch vendor of Enterprise Resource Planning (ERP) 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, as described Schunselaar et al. (2016)) will be expressive enough to fully describe the real-world enterprise of virtually any customer.

When a priority is assigned to the quality characteristics, (this is done with percentages, adding up to a total of 100%), the total score of both code generation and model interpretation can be calculated with the following formulas:

$$\sum_{i=1}^{j} P_i G_i \text{ and } \sum_{i=1}^{j} P_i I_i$$

For every quality characteristic j the priority is applied to either the code generation or interpretation preference. These values are summarized to end up with the total score of code generation and model interpretation.

Calculating the priorities of the different quality characteristics can be done in different ways, and this section describes two of them. In this case study observations are taken from two time-frames in the development process. Although the software development is a multi-year project and is still ongoing, these two time-frames represent two discussions on designing model execution approaches. The first time-frame focused on building an initial working version that could be used for exploration and validation. It is labeled the *Architecture Design Phase* as it focused on the initial architecture of the MDD platform. The second phase builds on top of the first phase, making the platform better suited for realistic deployment scenarios. Both phases are described in terms of the decisions that led the development, along with the requirements in terms of quality characteristics. Decisions that gave direction to the design are labeled with a **D**, and they are summarized in Table 2.

While it is observed that some of the decisions can be seen as requirements, it are actually the decisions following from corresponding requirements. For instance decision **D1** states that the decision is made to develop a MDD platform based on an OEM, following from the requirement to have a model with a high level of abstraction, not describing technology or software but organizations.

Table 2: Summary of the decisions.

Architecture Design Phase
D1 A MDD platform based on an OEM
D2 A SaaS delivery model for the application
D3 Use multi-tenancy to gain resource sharing
Deployment Design Phase
D4 Enable customers to customize the model
D5 Run the application on the .NET runtime
D6 Deploy as a distributed application
D7 Re-design the model execution for messages

4.1 Architecture Design Phase

Phase one focused on three key decisions, the first being the type of model used for the platform (**D1**). The foundational vision is that the model describes an organization, and not a software system. This results in the model being an OEM as described by Schunselaar et al. (2016). By definition this model has a high abstraction level, and lacks the details that are involved in creating software, details that are instead present in the software (generator or interpreter) transforming the model. The second decision was that the resulting application is delivered through the Software-as-a-Service (SaaS) model (**D2**). Within a SaaS model SPOs are paid for delivering a service, and it is important that this delivery is done in a cost effective way. Multi-tenancy is a manner for SPOs to be cost effective as stated in the definition of Kabbedijk et al. (2015). Therefore, the last decision followed naturally from the second: the platform would be multi-tenant (**D3**), although it was not yet decided what form of multi-tenancy. These three decisions gave focus for the development that was done in this timeframe. The SaaS model, along with the multi-tenancy resulted in a priority for the quality characteristics *resource utilization* and *run-time behavior*. The quality characteristic *confidentially* has no priority, because even if the model is distributed along with the software it will stay confidential in the SaaS model. The resources of the platform are shared among the customers, and to be cost effective, this sharing has to be optimal. The platform with a different model per customer also demanded confidence, so the development team gave priority to *testability*.

Although the data from this paper was not yet available, the decisions and requirements are in hindsight translated into a prioritization of the quality attributes as shown in Table 3. These weights are not used to validate the design against the knowledge from Table 1, because this study was done after this phase. They serve as an illustration of how to the data from the literature study can be used. Three out of four requirements show a clear preference for the code generation approach, leading to a 74% preference for code generation. Although according to the literature study testability and analysability suffer in the generation approach, the team was able to solve those with satisfaction. In this case, the experience confirmed the statements of Voelter (2009) and Voelter and Visser (2011) that code generation is easier to implement and debug. Phase one was indeed development by means of a full code generation approach.

Table 3: Summary of the prioritization of quality characteristics for phase one. Columns *G* and *I* show the preferences for code generation and model interpretation.

	Priority	G	I
Run-time behavior	0.35	0.88	0.12
Resource utilization	0.35	0.875	0.125
Testability	0.15	0.60	0.40
Analysability	0.15	0.222	0.778
Preference		0.738	0.262

4.2 Deployment Design Phase

Phase two took place some time after the first, and its goal was a version of the platform that would deal with the challenges of being a distributed SaaS solution that allows customers to maintain their own model. The decisions from phase one were not rejected, but new decisions were added to the list. The first was the possibility for customers to maintain and customize their own model (**D4**). This functionality requires an efficient upgrade mechanism, one which is not disruptive for the users. The team also decided on running the software on the .NET runtime (**D5**), a platform that does not support dynamic software updating, or other means of online upgrading. Dynamic software updating, or online upgrading, is a solution for updating software processes without stopping them first, a solution that results in an efficient upgrade mechanism. The quality characteristic *build-time behavior* became more important, because customers maintain their own model expecting near instant model execution, but the chosen platform (.NET) does not support updates without restarting the processes. The last decision was an extension of **D2**, the application should have a distributed nature to give robustness and optimal resource sharing (**D6**). The attributes *adaptability* and *modifiability* gained importance, because of the size of the platform and its source code.

To set boundaries in this phase, the model execution approach of a specific component was redesigned. The application has a distributed nature, and messages are used to pass information between

the different components. The re-design of the model execution approach for these messages was the objective for this phase (**D7**).

In this phase, the AHP method described by Saaty (1990) was used to rank the complete list of twelve quality characteristics. Falessi et al. (2011) conclude 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: as stated before there can be much detail in the model execution approach and the decision support method should support this. The second difficulty is that there are many quality attributes that need to be prioritized, and many attributes have small and subtle differences.

Table 4 shows the result of ranking the characteristics along with the total score for both code generation and model interpretation. The ranking is a result from the pairwise ranking of the characteristics in Table 1: every pair was ranked on relative importance. The attributes *build-time behavior*, *adaptability*, and *modifiability* have high priority (28%, 15.5%, and 15% respectively), based on the decisions made. Compared to phase one, the top three is completely different (see Table 3), and the preference is also flipped in favor of interpretation with 69%. The re-design was done by implementing a model simplification execution approach: the OEM is simplified into simple message contract definitions by the generator. Did *run-time behavior*, *resource utilization*, *testability*, and *analysability* became less important? No, but the decisions made in phase two favored other characteristics, such as *build-time behavior*, and initial tests showed that the simplification approach would not lead to an unacceptable decrease of the run-time performance or an unacceptable increase of the resource utilization. The team was able to satisfy the requirements for *run-time behavior*, *resource utilization*, *testability*, and *analysability* with the simplification approach, while *build-time behavior* was difficult to satisfy with the generative approach combined with the .NET runtime.

4.3 Evaluating the Decisions

The decisions that were observed during the design of the model execution approach are summarized in Table 2. As the result of the result of choices that were made in the design of the platform, these decisions set the boundaries and requirements for the ongoing design. The observed decisions are categorized in three areas *meta-model*, *architecture*, and *platform* and together form the context of the model execution approaches within a MDD platform.

Table 4: Summary of the prioritization of quality characteristics for phase two. Columns *G* and *I* show the preferences for code generation and model interpretation.

	Priority	G	I
Run-time behavior	0.059	0.88	0.12
Build-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	0.222	0.778
Modifiability	0.150	0.15	0.85
Testability	0.085	0.60	0.40
Adaptability	0.155	0.375	0.625
Installability	0.021	0.571	0.429
Preference		0.310	0.690

meta-model - Decisions **D1: OEM** and **D4: Customized model** shows how the meta-model and its features influence the design of the best fitting strategy for model execution. A meta-model with a high level of abstraction, results in a more complex model execution, because the high level of abstraction needs to be transformed into a running application. This complex model execution requires more resources and takes more time, therefore code generation, or model simplification is preferred. By implementing a transformation outside of the application (by either generating code, or simplifying the model), the required resources and processing time do not add any overhead to the application.

Decision **D4:Customize**, however, increases the priority of *build-time behavior*. Customers that change the model expect fast turn-around times, a quality characteristic that is better satisfied with model interpretation.

architecture - The multi-tenancy level in the architecture as defined by Kabbedijk et al. (2015) is also of influence to the model execution approach (see **D3: Multi-tenancy**). If an *application instance level* of multi-tenancy is required, code generation would not make sense unless the platform supports loading and unloading of source code. An application instance level of multi-tenancy uses a single process to serve different customers. Different customers can have different models, and the process should thus support loading (and unloading for efficient resource usage) of different sets of code for different customers. When multiple customers are served from the same platform, *resource utilization* becomes important. To be cost effective the platform maximizes resource sharing, but model interpretation adds to the overall resource usage.

Decisions **D6: Distributed application** and

D7: Re-design messages show how componentization gives engineers the possibility to apply different model execution approaches to different components in the software system. An application that consists of multiple loosely coupled services can implement a different model execution approach in every service. This approach can be extended to applications that run in a single process, but still consists of several loosely coupled components. When a component, that is tightly coupled to a generated component, uses model interpretation it might be necessary to regenerate the first component too, which then reverts any advantage of the interpretative approach. Examples for components that could benefit from a generative approach are those with more business logic or algorithms, where efficient code is important.

platform - Kelly and Tolvanen (2008) make no distinction between the architecture, framework, the operating system, or the runtime environment. The influence of the architecture and framework described in the previous area, are different from the operating system or runtime environment. As decision **D5: .NET platform** illustrates, a target platform that does not support dynamic software updating requires a different model execution approach to satisfy the build-time requirements. Not only phase two of the case study illustrates this, it is also seen in the approach taken by Meijler et al. (2010) with their customized Java class loader and with Czarnecki et al. (2002) using the extension object pattern.

Deciding for a **D2: SaaS delivery model** removes the priority from the characteristics *installability* and *co-existence*: the platform is controlled by the software company.

The case study has an illustrative nature, observations were made during two distinct phases of the design of an model execution approach. These phases are characterized by the different decisions that form the boundaries of the design. By showing two different situations, the influences from the context on the design of a model execution approach are shown. The two phases are related to the quality characteristics described in Section 3, and combined with observations on the decisions that were made in the design of the MDD platform. The resulting Tables 3 and 4 show how the different decisions result in different priorities for the two model execution approaches. In the first phase the decisions led to a preference for code generation by 74% versus 26% for model interpretation. Model interpretation received 69% preference in the second phase, because the shifted requirements led to different priorities for the quality characteristics.

Table 1 was received with mixed reactions by the development team. On the one hand the quality characteristics and the contextual influences guided the discussions and brainstorms by asking the questions that the team did not knew they needed to ask. The percentages, however, were not unanimously accepted. Questions were asked on the validity, mostly because the context of specific literature was doubted to be equal to the case study context. This demonstrates the importance of context in designing a model execution approach. It also illustrates the problems in transferring experience and knowledge from one context to another. It was concluded that the knowledge presented in Table 1 is informative, but the real value is in the experience behind the knowledge that triggers the “right” questions.

The case company decided to generate simplified models that are further interpreted at run-time. This showed a promising improvement in *build-time behavior* because there is less C# that needs to be generated and compiled. Interpreting parts of the model that influence the distributed nature of the application appears to be more difficult than generating it. With the results of this exploration, the team will continue to look out for components that benefit from a simplification approach, and the model execution approach will shift more towards a hybrid form of code generation and model simplification. A pure form of interpretation is considered to add too much overhead to the run-time performance of the application, because of the high level of abstraction that the OEM has.

Although this paper does not present a decision-making method, and it does not relieve SPOs from the hard work that designing a model execution approach is, it does make the knowledge and experience of relevant research accessible. By identifying the context of a model execution approach through decisions made in earlier phases, and by prioritizing the different quality attributes, a SPO can get insight in how well code generation and model interpretation satisfy the requirements. A result without a significant preference for either code generation or model interpretation steers the SPO into the design of a hybrid approach. Based on such a result, an hybrid approach helps optimizing the software quality by means of a better fitting model execution approach.

5 THREATS TO VALIDITY

The validity of this research is threatened by several factors. The construct validity, which is threatened by the fact that some of the researchers where involved in the object of the study: the observations made in section 4 could be biased. This threat is ad-

dressed by the fact that this paper is reviewed and commented on by key team members involved in the design of the execution approach, making sure that the observed decisions are correctly described.

The internal validity is threatened because the quality characteristics and the platform context on the one hand and the execution approach on the other hand are difficult to correlate. However, the claims made in the reviewed literature do converge towards each other. Although some characteristics lack a significant number of references, the authors regard the claims made in this paper as not being controversial, but in line with existing research. The data found in literature to support claims on quality lacks significance on a number of characteristics. Further, much of the literature that was used in composing the overview uses anecdotal argumentation, as it is based on the experience of the authors. The claims that were found in the cited work were frequently not validated in other cases, and no empirical evidence for the claims was given. To create a more trustworthy decision support method, the data presented in Table 1 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 case study done at a single company threatens the external validity. The consequences of the decisions made in the design process are in line with literature, but additional case studies should further strengthen the decision support method.

6 CONCLUSION AND FUTURE WORK

This paper makes two contributions to the research on MDD. The first contribution is the literature study and the resulting table with the preference for model execution approach per quality characteristic, described in Section 3. It makes years of experience and knowledge from many different authors on designing model execution approaches accessible. A summary is given with advantages and disadvantages that others can use in designing a fitting model execution approach. Although the knowledge was already available, it was scattered over many papers and hidden away as side-notes. With the overview of model execution approaches and quality criteria the paper provides SPOs with a method for rapid decision making when it comes to the question of interpretation versus generation.

The second contribution is presented in Section 4: an illustration of how the quality model for MDD platforms can be applied in a SPO designing a model execution approach. By giving priority to the quality characteristics, a SPO is forced to specify the requirements for the approach. The explicit listing of decisions that influence the model execution approach such as its architecture, platform choices, and its meta-model show how they interplay with the model execution approach. Some decisions, such as the abstraction level of the model, make a model execution approach as interpretation less preferable. Decisions such as the choice for a SaaS delivery model increase the need for interoperability and co-existence, increasing the preference for interpretation. This case study shows how the different quality characteristics influence the design, and may even conflict at different design phases of a system.

This paper is a first exploration of the design of a model execution approach. Although there exists a significant body of knowledge on MDD, including in-depth showcases of implementations of model execution approaches, none explores the designing of the most fitting approach for a given context. The exploration presented in this paper uncovers the need for more empirical research to validate the claims made in this paper on quality attributes. For instance the claim of performance of a generative approach over an interpretive approach is still widespread, but the actual overhead in a certain context might be negligible. Experimentation and large scale case studies should be conducted to build up evidence for the presented quality attributes. Not only the quality attributes and their preferences need more evidence. Studies on the design process should be done to improve the knowledge on how the design of a model execution approach evolves. Research should be done to define what the most fitting model execution approach entails, when is an approach fitting, and when is it better fitting than another approach.

Many, maybe all, questions in software development can be answered with *"it depends"*, leaving the questioner puzzled as to what he should do. Although this paper presents a first exploration and does not remove the difficulty from the designing and developing of a MDD platform, it does shed light on the design process. SPOs can use this exploration to improve their knowledge and 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 of this paper. Finally we thank the developers of AFAS Software for sharing their opinions and giving feedback.

REFERENCES

Batouta, Z. I., Dehbi, R., Talea, M., and Hajoui, O. (2015). Multi-criteria Analysis and Advanced Comparative Study Between Automatic Generation Approaches in Software Engineering. *Journal of Theoretical and Applied Information Technology*, 81(3):609–620.

Brady, E. C. and Hammond, K. (2010). Scrapping your inefficient engine. *ACM SIGPLAN Notices*, 45(9):297.

Capilla, R., Rey, U., Carlos, J., Dueñas, J. C., and Madrid, U. P. D. (2009). The Decision View's Role in Software Architecture Practice. *Practice*, March/April:36–43.

Cleenewerck, T. (2007). *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universteit Brussel.

Consel, C. and Marlet, R. (1998). Architecturing Software Using A Methodology For Language Development. *Principles Of Declarative Programming*, 1490(October):170–194.

Cook, W. R., Delaware, B., Finsterbusch, T., Ibrahim, A., and Wiedermann, B. (2008). Model transformation by partial evaluation of model interpreters. Technical report, Technical Report TR-09-09, UT Austin Department of Computer Science.

Cook, W. R., Delaware, B., Finsterbusch, T., Ibrahim, A., and Wiedermann, B. (2009). Strategic programming by model interpretation and partial evaluation. *unpublished*.

Cordy, J. R. (2004). TXLA language for programming language tools and applications. *In Proceedings of the ACM 4th International Workshop on Language Descriptions, Tools and Applications*, pages 1–27.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: Methods, Tools, and Applications*. Addison-Wesley Professional.

Czarnecki, K., Østerbye, K., and Völter, M. (2002). Generative programming. *Object-Oriented Technology, Proceedings*, 2323:135–149.

Díaz, V. G., Valdez, E. R. N., Espada, J. P., Bustelo, b. C. P. G., Lovelle, J. M. C., and Marín, C. E. M. (2014). A brief introduction to model-driven engineering. *Tecnura*, 18(40):127–142.

Ertl, M. A. and Gregg, D. (2003). The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25.

Fabry, J., Dinkelaker, T., Noye, J., and Tanter, E. (2015). A Taxonomy of Domain-Specific Aspect Languages. *ACM Computing Surveys*, 47(3):1–44.

Falessi, D., Cantone, G., Kazman, R., and Kruchten, P. (2011). Decision-making techniques for software architecture design. *ACM Computing Surveys*, 43(4):1–28.

Gaouar, L., Benamar, A., and Bendimerad, F. T. (2015). Model Driven Approaches to Cross Platform Mobile Development. *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, pages 19:1—-19:5.

Gregg, D. and Ertl, M. A. (2004). A Language and Tool for Generating Efficient Virtual Machine Interpreters. In *Domain-Specific Program Generation*, pages 196–215. Springer Berlin Heidelberg.

Guana, V. and Stroulia, E. (2015). How Do Developers Solve Software-engineering Tasks on Model-based Code Generators ? An Empirical Study Design. *First International Workshop on Human Factors in Modeling*, (May):33–38.

Hailpern, B. and Tarr, P. (2006). Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461.

Hen-Tov, A., Lorenz, D. H., and Schachter, L. (2008). ModelTalk: A Framework for Developing Domain Specific Executable Models. *The 8th OOPSLA Workshop on Domain-Specific Modeling*, (926):7.

Inostroza, P. and Van Der Storm, T. (2015). Modular interpreters for the masses implicit context propagation using object algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 171–180. ACM.

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.

Jansen, A. and Bosch, J. (2005). Software Architecture as a Set of Architectural Design Decisions. *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120.

Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International.

Jörges, S. (2013). *Construction and evolution of code generators: A model-driven and service-oriented approach*, volume 7747. Springer.

Kabbedijk, J., Bezemer, C.-P., Jansen, S., and Zaidman, A. (2015). Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, 100:139–148.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling: enabling full code generation*. John Wiley & Sons.

Klint, P. (1981). Interpretation Techniques. *Software: Practice and Experience*, 11(June 1979):963–973.

Meijler, T. D., Nytun, J. P., Prinz, A., and Wortmann, H. (2010). Supporting fine-grained generative model-driven evolution. *Software & Systems Modeling*, 9(3):403–424.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344.

Ousterhout, J. K. (1998). Scripting: Higher-Level Programming for the 21st Century. *Computer*, 31(3):23–30.

Riehle, D., Fraleigh, S., Bucka-Lassen, D., and Omorogbe, N. (2001). The architecture of a UML virtual machine. *International Conference on Object Oriented Programming Systems Languages and Applications (OOSPLA)*, (February):327–341.

Rohou, E., Swamy, B. N., and Seznec, A. (2015). Branch prediction and the performance of interpreter - Don't trust folklore. *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 103–114.

Romer, T. H., Lee, D., Voelker, G. M., Wolman, A., Wong, W. a., Baer, J.-L., Bershad, B. N., and Levy, H. M. (1996). The structure and performance of interpreters. *ACM SIGPLAN Notices*, 31(9):150–159.

Saaty, T. (1990). How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1):9–26.

Schramm, A., Preußner, A., Heinrich, M., and Vogel, L. (2010). Rapid UI development for enterprise applications: Combining manual and model-driven techniques. *Models*, 6394 LNCS(PART 1):271–285.

Schunselaar, D. M. M., Gulden, J., Van Der Schuur, H., and Reijers, H. A. (2016). A Systematic Evaluation of Enterprise Modelling Approaches on Their Applicability to Automatically Generate Software. In *18th IEEE Conference on Business Informatics*.

Shali, A. and Cook, W. R. (2011). Hybrid Partial Evaluation. *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 375–390.

Smolik, P. and Vitkovsky, P. (2012). Code Generation Nirvana. *Modelling Foundations and Applications*, pages 319–327.

Stahl, T., Völter, M., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*.

Svahnberg, M., Wohlin, C., Lundberg, L., and Mattsson, M. (2003). A Quality-Driven Decision-Support Method for Identifying Software Architecture Candidates. *International Journal of Software Engineering and Knowledge Engineering*, 13(05):547–573.

Tankovic, N. (unknown). Model Driven Development Approaches: Comparison and Opportunities. Technical report.

Tanković, N., Vukotić, D., and Žagar, M. (2012). Rethinking Model Driven Development : Analysis and Opportunities. *Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference*, pages 505–510.

Thibault, S. and Consel, C. (1997). A framework for application generator design. *ACM SIGSOFT Software Engineering Notes*, 22(3, May 1997):131–135.

Thibault, S. A., Marlet, R., and Consel, C. (1999). Domain-Specific Languages : From Design to Implementation Application to Video Device Drivers Generation. *IEEE Transactions on Software Engineering*, 25(3):363–377.

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.

Varró, G., Anjorin, A., and Schürr, A. (2012). Unification of compiled and interpreter-based pattern matching techniques. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7349 LNCS:368–383.

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

Voelter, M. and Visser, E. (2011). Product Line Engineering Using Domain-Specific Languages. *15th International Software Product Line Conference*, (Section II):70–79.

Wohlin, C. (2014). Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. *18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*, pages 1–10.

Yoder, J. W. and Johnson, R. (2002). The Adaptive Object-Model Architectural Style. *The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)*, pages 1–25.

Zhu, L., Uke, a. Y. B., Gorton, I. a. N., and Jeffery, R. (2005). Tradeoff and Sensitivity Analysis in Software Architecture Evaluation Using Analytic Hierarchy Process. pages 357–375.

Zhu, M. (2014). *Model-Driven Game Development Addressing Architectural Diversity and Game Engine-Integration*. PhD thesis, Norwegian University of Science and Technology.