MARCO CRAVEIRO

NOTES ON MODEL DRIVEN ENGINEERING

NOTES ON MODEL DRIVEN ENGINEERING

MARCO CRAVEIRO

A Literature Review
University of Hertfordshire

December 2021 — v1.0

Marco Craveiro: *Notes on Model Driven Engineering* A Literature Review
© 2021

LOCATION:
Hatfield

TIME FRAME:
December 2021

Malembe malembe. — old Angolan proverb.

ABSTRACT

Model Driven Engineering (MDE) is an established approach for the engineering of software systems. MDE distinguishes itself from related approaches due to its explicit focus on the *modeling* of entities within a software system, and on the *transformations* that can be applied to them.

The present manuscript provides a brief introduction to key MDE concepts, with an associated critique where applicable, and discusses its interaction with two important aspects of the software engineering discipline: the software development process itself, and the modeling of variability within a software product.

The target audience for this work are experienced software engineers with little to no knowledge of model driven techniques, and practitioners who wish to revisit the fundamentals.

CONTENTS

LIST OF FIGURES

ACRONYMS

AC-MDSD	Architecture-Centric MDSD.
AO	Aspect Oriented.
AO-MD-PLE	Aspect-Oriented Model Driven PLE.
AOP	Aspect Oriented Programming.
API	Application Programming Interface.
ATL	Atlas Transformation Language.
BDD	Binary Decision Diagram.
CLR	Common Language Runtime.
CNF	Conjunctive Normal Form.
CVL	Common Variability Language.
DOM	Document Object Model.
DSL	Domain Specific Language.
EMF	Eclipse Modeling Framework.
FODA	Feature-Oriented Domain Analysis.
FOP	Feature-Oriented Programming.
GMF	Graphical Modeling Framework.
GP	Generative Programming.
GPML	General Purpose Modeling Language.
IDE	Integrated Development Environment.
IPC	Implicit Presence Conditions.
J2EE	Java Platform Enterprise Edition.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
LDL	Layer Definition Language.
M2M	Model-to-Model.
M2T	Model-to-Text.
MASD	Model Assisted Software Development.
MBE	Model Based Engineering.
MBSE	Model-based Systems Engineering.
MDA	Model Driven Architecture.
MDD	Model Driven Development.
MDE	Model Driven Engineering.
MDSD	Model Driven Software Development.
MDSE	Model Driven Software Engineering.
MIC	Model-Integrated Computing.
MOP	Model Oriented Programming.

MT	Model Transformation.
OMG	Object Management Group.
OO	Object-Oriented.
OVM	Orthogonal Variability Modeling.
PCL	Product Configuration Language.
PDM	Platform Description Model.
PIM	Platform Independent Model.
PM	Physical Model.
PSM	Platform Specific Model.
QVT	Query / View/ Transformation.
RTE	Round-Trip Engineering.
SAX	Simple API for XML.
SDLC	Software Development Lifecycle.
SDM	Software Development Methodology.
SPLE	Software Product Line Engineering.
T2T	Text-to-Text.
TS	Technical Space.
UML	Unified Modeling Language.
XML	Extensible Markup Language.
XSD	XML Schema Definition Language.

Part I

FUNDAMENTALS

1 INTRODUCTION

Better integration of such models and code should significantly increase the opportunity to effect change through the models, rather than simply modifying the code directly.

— Krzysztof Czarnecki [Völ+13]

Model Driven Engineering (MDE) is an approach to the development of software systems based on modeling and transformations. The present monograph provides a succinct characterisation of key aspects of MDE, and discusses its interaction with both the software development process and the management of variability. The analysis here presented was carried out in the spirit of an academic literature review, with the purpose of sketching out basic boundaries across the vast field of MDE.

Target audience

The present work is best understood as the *Companion Notes* or *preamble* of our doctoral dissertation; its objective is to supply the basic building blocks required by those unfamiliar with modeling technologies to fully understand the argument there put forward.[1] In addition, the material should also be suitable to experienced software engineers who need an introduction to MDE, such as developers wanting to participate in the development of Dogen — a software project led by the author of this document.[2,3] Finally, experienced MDE practitioners may also be served by the manuscript if they wish to revisit and critique some of the discipline's foundational concepts.

Organisation

The work is divided into three parts, and it is organised as follows:

- **Part i: Fundamentals**: Tackles key theoretical aspects of the discipline of MDE. It is composed of the following chapters:
  - **Chapter 1: Introduction** — The present chapter, providing an overview of the manuscript.
  - **Chapter 2: Towards a definition of MDE** — Attempts to characterise MDE according to commonly used terms in computer science such as paradigm, methodology and others of the same ilk.

1 Whilst the material here presented is extremely relevant to our analysis of MDE's theoretical framework, it was deemed to have insufficient novelty when compared to the field's state of the art — feel as we might that these chapters make important points not elsewhere made.
2 https://github.com/MASD-Project/dogen
3 Those requiring a more thorough introduction to the subject matter are guided towards [BCW12] and [Völ+13].

  - **Chapter 3: Models and Transformations** — Provides a short overview of the two most important concepts in MDE, *models* and *transformations*; describes the modeling activity, and relates modeling languages to programming languages.

  - **Chapter 4: From Problem Space to Solution Space** — Introduces the problem space and the solution space, and discusses their relationship with MDE. Relates these concepts to *platforms* and *Technical Spaces (TSs)*.

- **Part ii: Integrations**: Discusses the integration of MDE with related software engineering concepts. It is comprised of the following chapters:

  - **Chapter 5: MDE and the Software Development Process** — Places MDE in the context of the typical software development process, broaching the challenges of this integration.

  - **Chapter 6: MDE and Variability Modeling** — Provides an overview of variability modeling, and supplies many of the primitives needed to contextualise variability management in the presence of MDE.

- **Part iii: Outlook**: Contains a single chapter with our conclusions:

  - **Chapter 7: Conclusion** — Summarises the present work and discusses points for further study.

2 TOWARDS A DEFINITION OF MDE

It is striking to see however, that though MDE is supposed to be about precise modelling, MDE core concepts are not defined through precise models. [...] In such conditions, it is very difficult to understand how the basic concepts of MDE are related, and how they can be mapped to existing Technological Spaces.

— Jean-Marie Favre [Fav04]

THE FIRST CHALLENGE facing a new MDE practitioner is in obtaining a rigorous definition of the discipline and its boundaries. The present chapter aims to guide the reader towards an understanding of the discipline, across a difficult landscape.

Chapter overview

The chapter is organised as follows. First, Section 2.1 will get our bearings by positioning MDE against a number of computer science terms such as *paradigm* and *methodology*. Section 2.2 will then home in on the aspects related to a *body of knowledge*, and it is followed by a discussion of the hierarchical model of modeling approaches (Section 2.3). We will then discuss the complexities associated with the vast number of acronyms in the field (Section 2.4), sketch out the boundaries of the discipline (Section 2.5) and understand the role of pragmatism in modeling (Section 2.6). The chapter concludes with a brief discussion outlining our views on the role of MDE (Section 2.7).

2.1 WHAT, IF ANYTHING, IS MDE?

Defining the nature of MDE is not quite as simple as it may appear. On one hand, informal characterisations abound, with most similar in spirit to what Cuesta proposes (*emphasis ours*):

Informal characterisation

> [Within MDE, software] development processes are conceived as *a series of steps* in which *specification models*, as well as those which describe the problem domain, are *continually refined*, until implementation domain models are reached — along with those which make up the verification and validation of each model, and the correspondence between them. In [MDE], the steps in the development process are considered to be *mere transformations between models*.[1,2] [Cue16] (p. 1)

1 This quote was translated from the original Spanish by the author. The reader is advised to consult the primary source.

2 The nature of models within MDE, as well as that of transformations, is analysed in greater detail on Chapter 3.

On the other hand, formal definitions are found wanting. To make matters worse, the literature remains unclear as to whether MDE is an approach [Fav04], a methodology [BCW12], a technical space [Béz05a], a paradigm [Völ+13; GHN10], all, some or neither of these. Furthermore, there is a degree of imprecision as to how most of these terms are generally used within computer science. As an example, Harper made known his dissatisfaction with the term *paradigm* in his essay "What, if anything, is a programming paradigm?", stating:

Unclear status, lax terminology

> The trouble with programming paradigms is that it is rather difficult to say what one is, or how we know we have a new one. Without precise definitions, nothing precise can be said, and no conclusions can be drawn, either retrospectively or prospectively, about language design. [Har17]

Programming paradigms

MDE literature steers into similarly difficult terrain. In [BCW12], Brambilla *et al.* call MDE a methodology but then hasten to qualify they (*emphasis ours*)

> [...] realise that *methodology* is an overloaded term, which may have several interpretations. In this context we are not using the term to refer to a formal development process, but instead as *a set of instruments and guidelines* [...]. [BCW12] (p. 7)

Those qualifications and the imprecision of the language raise concerns as to the usefulness of the term, so we choose instead to side with Matinnejad [Mat11] as well as Asadi and Ramsin [AR08], who make a strong case for calling MDE an *approach* or a *general framework* rather than a methodology.[3] With regards to the remaining terms: heeding Harper's warnings above, we feel one should refrain from using the term paradigm altogether; and, whilst agreeing with the notion of MDE as a *technical space*, we address it in what we believe to be a more fitting context (*cf.* Section 4.2.1).

Approach, general framework

The matter does not rest here, though, for there are additional viewpoints that must be taken into account when attempting to characterise MDE.

2.2 MDE AS A "BODY OF KNOWLEDGE"

In our opinion, MDE is *more* than an approach. For our purposes, MDE is to be understood as a large *body of knowledge* focused on the employment of modeling as the principal driver of software engineering activities. This choice of wording may appear controversial at first sight, given the position of Mussbacher *et al.* [Mus+14] — who stated that "[unlike] most other fields of engineering, model-driven engineering does not have a Body of Knowledge (BoK) as such." Nevertheless, both views can be reconciled by accepting a distinction between a formal *Body of Knowledge* — which MDE lacks — and an informal *body of knowledge* — which we posit MDE *is*.

Body of knowledge

3 This matter is argued further on Section 5.1, where we address the relationship between MDE and the software development process, and articulate a stronger argument as to why MDE should not be considered to be a methodology. Furthermore, we also provide additional context on software development methodologies.

The idea can perhaps be clarified by recourse to the words of Fairley and Chaffer (*emphasis ours*):

> Every profession is based on *a body of knowledge*, although that knowledge is *not always defined in a concise manner*. In cases where no formality exists, the body of knowledge is *"generally recognized"* by practitioners and may be codified in a variety of ways for a variety of different uses. [BF+14]

Informal body of knowledge

This, we believe, is an apt description of MDE from the perspective of the MDE practitioner. But in order for a practitioner to make sense of this vast and diverse body of knowledge, it is useful to arrange it into some form of structure.

2.3 a hierarchy of modeling approaches

One common method of overlaying structure into MDE is by way of classifying the application of modeling within the software development process. Here, researchers often distinguish between three different "levels" of modeling use, with each level seen as a super-set of the preceding [Ame; WHR14; BCW12]. At the lowest level sits Model Based Engineering (MBE), where models are employed as a form of inter and intra-team communication and documentation. Within MBE, models are an input to the development process but, typically, software engineers are responsible for manually translating them into source code.

Hierarchy of modeling approaches, MBE

Next, at an intermediate level, sits Model Driven Development (MDD), where the software development process is driven entirely by models, and whose translation to code is performed by automated means. Finally, MDE is located at the highest level of the hierarchy; it orchestrates *all* engineering activities of a software system via modeling, including its development, operation and maintenance.[4] Figure 2.1 provides a graphical illustration of these relationships.

MDD, MDE

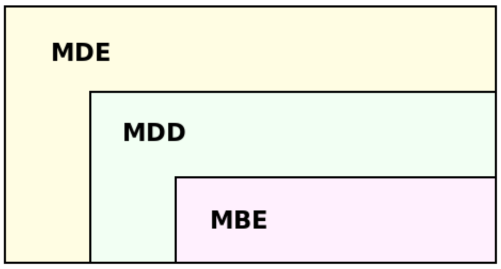


Figure 2.1: Hierarchy of modeling approaches. *Source*: Author's drawing based on a diagram by Whittle *et al.* [WHR14]

Whilst this hierarchy is conceptually convenient and appeals to the intuition of most software engineers — possibly because it echoes earlier notions

4 Here we resort to Bourque *et al.*'s definition of Software Engineering: "[. . .] the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software." [Abr+04]

around Object-Based and OO programming[5] — in practice, the layers are notoriously difficult to tease apart and are thus of limited use outside introductory material. *Practical difficulties* Indeed, empirical observations such as those of Hutchinson *et al.* [Hut+11] reveal that they are closer to a spectrum of possibilities rather than a set of well-defined categories. As a result, the terms Model Driven Development (MDD) and MDE are used interchangeably in much of the literature, and their presence often merely signifies "*a* model-driven approach" instead of the more technical meaning intended by the hierarchy. And, sadly, this is just the end *of the beginning* of the confusing terminology.

2.4 THE MODEL-DRIVEN JUNGLE

Matters are further complicated by the existence of a number of additional model-driven approaches and methodologies outside the hierarchy, similar in characteristics but differing in naming, such as Model Driven Software Engineering (MDSE) [BCW12], Model Driven Software Development (MDSD) [SVC06], Model Driven Architecture (MDA) [Poo01], Model-Integrated Computing (MIC) [Spr04], Model Oriented Programming (MOP) [BL13], Model-based Systems Engineering (MBSE) [Est+07] and many others. *Variants, flavours* The literature customarily refers to these as *MDE variants* [BCW12] (p. 10) or *MDE flavours* [Völ+13] (p. 11), but these are not precise terms. Interestingly, our review did not uncover a much needed taxonomy or a detailed comparative study characterising variants and their relationships, in what appears to be a glaring gap in the literature.

The need for grouping is certainly present, for it is cumbersome have to name variants individually when making arguments that apply to *model-driven* in general. Völter sought to redress this shortcoming by grouping variants under the umbrella term *MD** [Völ09] and it quickly became established practice in the literature. *MD** However, it is not yet clear if the existence of this grouping has helped or hindered those seeking to understand what MDE *is*, specially as there is no rigorous definition of the members of the group nor of its properties, much beyond Völter's original — and uncharacteristically carefree — comment: "I use MD* as a common moniker for MDD, MDSD, MDE, MDA, MIC, and all the other abbreviations for basically the same approach." [Völ09]

With a hint of well-placed irony, Brambilia *et al.* spoke of the *MD* Jungle* [BCW12] (p. 9), adding that (*emphasis ours*) "MDE can be seen as the superset of all these variants, as *any* [MD*] approaches could fall under the MDE umbrella." [BCW12] (p. 10) *MDE as the superset* It is in this sense that the term MDE is to be understood within this dissertation — a sense, we believe, that is entirely consistent with its role as a body of knowledge. However, questions are

5 The adjectives *based*, *oriented*, and *driven* are rife throughout computer science and software engineering, obscuring somewhat their intent and differences. For example, Meyer is not entirely convinced about how *oriented* and *based* are used in the OO context (*emphasis his*): "Because the English words *based* and *oriented* do not readily evoke the conceptual difference between encapsulation techniques and OO languages, 'object-based' is a little hard to justify, especially to newcomers." [Mey88] (p. 1100) The modeling community largely circumvented such difficulties by giving preference to *driven* — *e.g. model-driven* — over the arguably more obvious *oriented* — *e.g.* "model-oriented". As Stahl *et al.* put it, "The adjective 'driven' in 'Model-Driven Software Development' — in contrast to 'based' — emphasizes that this paradigm assigns models a central and active role: they are at least as important as source code." [Völ+13] (p. 4)

then raised as to how best determine what is *inside* the informal body of knowledge, as opposed to what remains *outside*.

2.5 CONDITIONS AT THE BOUNDARIES

Much like the boundaries from within, the boundaries from without are no less troublesome to isolate. Over time, MDE has been integrated with a number of existing approaches and methodologies such as Software Product Line Engineering (SPLE) [PBDL05], Agile [Mat11], Generative Programming (GP) and Domain Engineering [Cza98] — to name but a few — to an extent that its now difficult to determine whether certain concepts should be included as part of MDE's body of knowledge or are extraneous to it. A similar problem occurs with terms defined in its variants such as Model Driven Architecture (MDA), particularly for those which have wider applicability to modeling problems outside the MDE variant itself.

Unclear boundaries

Our review of the literature did not uncover any adequate solutions to this thorny problem. Though by no means authoritative, our approach was to include in the present analysis all of the concepts that are relevant to our purposes and to use the notion of *integrations* where the concepts are more obviously external to MDE — *e.g.*, "Agile integration" (*cf.* Part ii). But all the difficulties discussed thus far point out there may be a deeper malaise with the discipline itself.

Integrations

2.6 PRAGMATISM IN A FUZZY DISCIPLINE

Jörges *et al.* encapsulated all of these themes in a manner that could be construed as an indictment to the entire endeavour (*emphasis ours*):

> This "fuzziness" or lack of precision can be observed for most of the vocabulary used in the context of MD*. *There is still no established fundamental theory of modeling* and related concepts that would be comparable to the maturity achieved in other disciplines of computer science, such as compiler construction. [Jör13] (p. 14)

Fuzziness, lack of precision

However, behind the lack of precision lies method, as Bézivin's incisively explains: "[we] are not interested here by a theoretical definition [. . .] but by an engineering one, *i.e.* a definition that will help users to implement and maintain systems." [Béz05a] Citing Fowler [Fow04], Brambilla *et al.* go as far as warning practitioners to "[b]eware of statements of pure principles: when it comes down to it, the real point of software engineering practice is to increase productivity, reduce errors, and cut code." [BCW12] (p. 23) Translated to the software development vernacular, MDE sees *fuzziness as a feature, not a bug*. This is a very significant statement of intent. For the

Need for pragmatism

remainder of this work we shall refer to it as the *Pragmatism Principle*, albeit restated in a slightly different form:[6]

Pragmatism Principle

> When defining terms within MDE's body of knowledge, *engineering definitions* are preferred over *theoretical definitions*. That is, pragmatic definitions that help practitioners implement and maintain the systems of today, even when imprecise, are preferred over rigorous theoretical definitions that are either not yet completely formulated or that fail to meet the helpfulness criteria.

Consequences of pragmatism

In our opinion, the Pragmatism Principle helps explain the apparent fuzziness of a significant subset of MDE's vocabulary for, in an environment where operational definitions abound, there is a permanent danger of duplication, inconsistencies and misunderstandings. After all, what suffices for one use case may not do at all for others. Nonetheless, we view the Pragmatism Principle as an important factor in MDE's progress and find it to be consistent with our view of MDE as a body of knowledge rather than a methodology or a paradigm, as we are no longer constrained by a need for rigour at all costs or even for overall consistency.

Importance of rigour

Note that the Pragmatism Principle is not a *carte blanche* to legitimise and empower sloppy reasoning. Rigour is still important to MDE, as are theoretical foundations, and the lack of fundamentals often appears in MDE research roadmaps [FR07; Mus+14]. The Pragmatism Principle merely justifies using an empiric approach to enable progress whilst the theoretic foundations are being laid, and presupposes the ability to make a trade-off between rigour and applicability where needed. And these trade-offs are not restricted to rigour and pragmatism either.

2.7 DISCUSSION

MDE as a supplier of building blocks

Seen from the present vantage point, MDE's role — *de facto* if not *de jure* — has been to provide the building blocks from whence model-driven methodologies can be constructed, to the precise specifications of its practitioners. And, in this regard, it has been very successful. In our opinion, the main downside of the Pragmatism Principle — and thus, of the approach as a whole — is that it places a *great deal of responsibility* on the practitioner to make the correct trade-offs, therefore requiring a *high-level of mastery* of a large and complex cannon.[7]

At this juncture we can now sketch out our understanding of the discipline, which is as follows:

6 The term *pragmatism* was chosen as an allusion to one of Stachowiak's fundamental model properties (*cf.* Section 3.1) because we see MDE *as a model too*. Alas, a detailed discussion of the topic would take us too far afield, befitting the philosophy of modeling in software development.

7 Take the decision to use modeling in the first place. Whilst the enumeration of choices presented in Section 3.7 appears to convey simplicity, experience says otherwise. In order to make an informed decision, one must first master *both* the theory of MDE *as well as* the thorny practical aspects of its application, and these only reveal themselves when applied to a *sufficiently large* project during *sufficiently long* timescales — as we ourselves discovered.

- MDE is an *informal body of knowledge* centred on the employment of modeling as the principal driver of software engineering activities.

- MDE promotes the *pragmatic application* of a *family of related approaches* to the development of software systems, with the intent of *generating automatically* a part or the totality of a software product, from one or more *formal models* and associated *transformations* (*cf.* Section 3).

Discipline characterisation

- MDE is best understood as a *vision* rather than a *concrete destination*. A vision *guides* the general direction of the approach, but does not dictate the solution, nor does it outline the series of steps required to reach it.[8]

- It is the responsibility of the *MDE practitioner* to select the appropriate tools and techniques from the MDE body of knowledge, in order to apply it adequately to a *specific instance* of the software development process. By doing so, the practitioner will create — *implicitly* or *explicitly* — an MDE variant.

Now that we have a basic understanding of MDE's reach, we can turn our attention towards its core concepts.

8 This vision is articulated clearly by France and Rumpe:

> In the MDE vision, domain architects will be able to produce domain specific application development environments (DSAEs) using what we will refer to as MDE technology frameworks. Software developers will use DSAEs to produce and evolve members of an application family. A DSAE consists of tools to create, evolve, analyze, and transform models to forms from which implementation, deployment and runtime artifacts can be generated. Models are stored in a repository that tracks relationships across modeled concepts and maintains metadata on the manipulations that are performed on models. [FR07]

3 MODELS AND TRANSFORMATIONS

Accordingly, all of cognition is cognition in models or by means of models*, and in general, any human encounter with the world needs "model" as the mediator [. . .].*

— Herbert Stachowiak [Pod17]

At the core of MDE sit two key concepts: that of *models* and of *transformations*. These notions are so central to MDE that Brambilia *et al.* distilled the entire discipline into the following equation [BCW12] (p. 8):

$$Models + Transformations = Software \tag{3.1}$$

Chapter overview

This chapter provides an overview of both of these concepts. It is organised as follows. It starts by determining the need for models in the first instance (Section 3.1), and then elaborates more precisely what is meant by the term *model* in this context (Section 3.2). Next, we analyse the relationship between models and metamodels (Section 3.3) as well as their hierarchical nature (Section 3.4). Following this, we provide a context for modeling languages, first by looking at their *purpose* (Section 3.6) and then by contrasting and comparing them with the better known programming languages (Section 3.8). The chapter ends with a brief analysis on how to approach modeling.

Let us begin then by considering the role of models in the general case.

3.1 WHY MODEL?

Sources

Unlike the difficulties faced in defining MDE's boundaries (*cf.* Section 2.5), the literature presents a far more consensual picture on the reasons for modeling. Predictably, it emanates mainly from without, given the importance of models in most scientific and engineering contexts. Due to this, we have limited our excursion to two often cited sources in the early MDE literature: Rothenberg and Stachowiak.

In "General Model Theory" [Sta73], Stachowiak proposes a model-based concept of cognition and identifies three principal features of models:

- **Mapping**: Models map individuals, original or artificial, to a category of all such individuals sharing similar properties. The object of the mapping could itself be a model, thus allowing for complex composition.

- **Reduction**: Models focus only on a subset of the individual's properties, ignoring aspects that are deemed irrelevant.

Model properties

- **Pragmatism**: Models have a purpose as defined by its creators, which guides the modeling process. Stachowiak states (*emphasis ours*): "[models] are not only models of something. They are also models for somebody, a human or an artificial model user. They perform therefore their functions in time, within a time interval. *And they are finally models for a definite purpose*."[1]

Much of the expressive power of models arises from these three fundamental properties.

For his part, Rothenberg's contribution [Rot+89] also gives a deep insight into the nature of models and the modeling process[2] and, in particular, his statements on substitutability are of keen interest when uncovering the reasons for modeling (*emphasis his*):

Substitutability

> Modeling, in the broadest sense, *is the cost-effective use of something in place of something else for some cognitive purpose*. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.

Models and software

Taken together, the properties identified by Rothenberg and Stachowiak make it clear that software engineering is inevitably deeply connected to models and modeling, as is any other human endeavour that involves cognition. However, this implicit understanding only scratches the surface of possibilities. In order to extract all of the potential of the modeling activity, *explicit* introspection is necessary. Evans eloquently explains why it must be so (*emphasis ours*):

Advantages of models

> To create software that is valuably involved in users' activities, a development team must bring to bear *a body of knowledge related to those activities*. The breadth of knowledge required can be daunting. The volume and complexity of information can be overwhelming. *Models are tools for grappling with this overhead*. A model is a *selectively simplified and consciously structured form of knowledge*. An *appropriate model* makes sense of information and focuses it on a problem. [Eva04] (p. 3)

1 The quote was sourced from Podnieks [Pod17] (p. 19). As we could not locate an English translation of "General Model Theory" [Sta73], we were forced to rely on secondary sources, including Podnieks, to access fragments of Stachowiak's work. Podnieks' paper is of great interest with regards to the philosophical aspects of modeling, but lays beyond the scope of our dissertation.

2 The paper is recommended reading to anyone with interest in the philosophical aspects of modeling and its relation to computer science. Readers are also directed to Czarenecki [Cza+00], Chapter 2 "Conceptual Modeling" and to Bézivin [Béz05b], Section 3.1. "On the meaning of models". Incidentally, it was Bézivin's paper that guided us towards Rothenberg's work.

Therefore, it is important to model *consciously*; with a purpose. In order to do so, one must first start with a better understanding of what is meant by "model" in the context of the target domain.

3.2 WHAT KIND OF MODEL?

The term *model* is used informally by software engineers as a shorthand for any kind of abstract representation of a system's function, behaviour or structure.[3] Ever critical in matters of rigour, Jörges *et al.* [Jör13] (p. 13) remind us that "[the] existence of MD* approaches and numerous corresponding tools [. . .] indicates that there seems to be at least a common intuition of what a model actually is. However, there is still no generally accepted definition of the term 'model'." In typical pragmatic form, Bézivin [Béz05a] provides what he calls an "operational engineering definition of a 'model'" (*emphasis ours*): "[. . .] a graph-based structure representing some aspects of a given system and *conforming to the definition of another graph* called a metamodel." Stahl *et al.* call this a *formal model* [SVC06] (p. 58).

Formal Model

The crux here is to address ambiguity. A formal model is one which is expressed using a formal language, called the *modeling language*, and designed specifically for the purpose of modeling well-defined aspects of a problem domain. A modeling language needs to be a formal language because it must have well-defined *syntax* and *semantics*, required in order to determine if its instances are well-formed or not. The next sections provide an overview of these two topics; since the literature was found to be largely consensual, the focus is solely on their exposition. We then join these two notions as we revisit the concept of metamodel.

Modeling Language

3.2.1 *Syntax*

Syntax concerns itself with *form*, defining the basic building blocks of the language and the set of rules that determine their valid combinations. This is done in two distinct dimensions:

- **Concrete Syntax**: Specifies a physical representation of the language, textually or graphically. It can be thought of its external representation or *notation*, of which there can be one or more.

Syntax types

- **Abstract Syntax**: Specifies the language's underlying structure, independent of its concrete syntax. It can be thought of as its internal representation.

3 An idea which is, in itself, a model taken from software design [AF16].

3.2.2 *Semantics*

Validation is not complete at the syntactic level, however, because a statement's validity may be dependent on context and therefore requiring an understanding of its *semantics*. Semantics deals with *meaning* and, as with syntax, it is also split across two distinct dimensions:

Semantics types

- **Static Semantics**: Concerns itself with contextual aspects that can be inferred from the abstract syntax representation of the model. In the case of a typed general purpose programming language, static semantics are comprised of a set of rules that determine if an expression is well-formed given the types involved. For modeling languages, the exact role of static semantics varies but is commonly concerned with placing constraints on types.

- **Dynamic Semantics**: These are only relevant to modeling languages whose instances can be executed and are thus also known as execution semantics. They define the execution behaviour of the various language constructs.

3.3 MODELS AND METAMODELS

Metamodel

Given these concepts, we can now elaborate further on Bezivin's definition above, and connect them from a modeling point of view. Formal models are instances of a modeling language, which provides the modeler with the vocabulary to describe entities from a domain. Together, the abstract syntax and the static semantics of the modeling language make up its *metamodel*, and instance models — by definition — must *conform* to it.

Structure

Employing terminology from Kottemann and Konsynsk [KK84], the metamodel can be said to capture the *deep structure* that connects all of its instance models, and the instance models are expressed in the abstract syntax of the modeling language — its *surface structure*. Within this construct, we now have a very clear separation between the entities being modeled, the model and the model's metamodel as they exist at different *layers of abstraction*.[4] However, the layering process does not end at the metamodel.

3.4 METAMODELLING HIERARCHY

Metametamodel

Since all formal models are instances of a metamodel, the metamodel itself is no exception: it too must conform to a *metametamodel*. The metametamodel provides a generalised way to talk about metamodels and exists at a layer above that of the metamodel. Though in theory infinite, the layering process

4 It is worth noticing that the use of metamodels in the context of OO languages has a long history in computer science. Henderson-Sellers *et al.* [HS+13], in their interesting and thought provoking work, report of their emergence within this context: "The use of metamodels for OO modelling languages was first promoted in 1994 [. . .] and consequently realized in Henderson-Sellers and Bulthuis [HSB12] in their creation of metamodels for 14 out of a list of 22 identified (then

is typically curtailed at the metametamodel layer, since it is possible to create a metametamodel that conforms to itself.[5],[6]

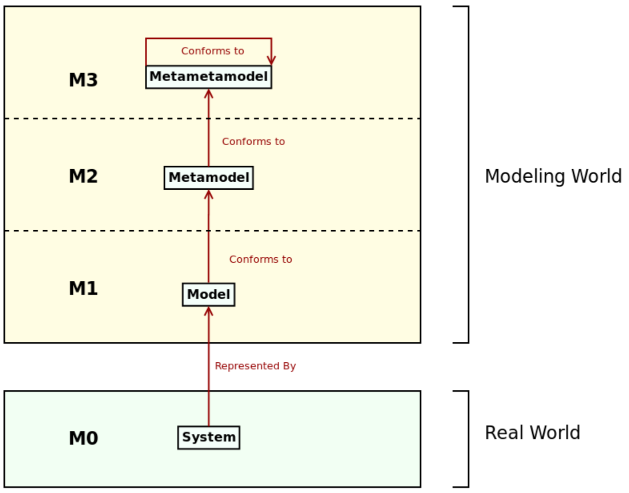


Figure 3.1: OMG four-layer metamodel architecture. *Source*: Author's drawing based on a diagram by Bézivin [Béz04].

Following on from the above-mentioned work of Kottemann and Konsynski [KK84], and that of many others, the Object Management Group (OMG) standardised these notions of an abstraction hierarchy into a *four-layer metamodel architecture* that describes higher-order modeling. Bézivin [Béz05b] referred to it as the *3+1 architecture*, and summarised it as follows: "[at] the bottom level, the M0 layer is the real system. A model represents this system at level M1. This model conforms to its metamodel defined at level M2 and the metamodel itself conforms to the metametamodel at level M3." Figure 3.1 illustrates the idea. Its worthwhile pointing out that the four-layer architecture is a typical example of the constant cross-pollination within MDE, as it was originally created in the context of what eventually became the Model Driven Architecture (MDA) but nowadays is seen as part of the core MDE cannon itself [BCW12].[7]

Four-layer metamodel architecture

As the literature traditionally explains the four layer model by means of an example [Béz05b; BCW12; HSB12], we shall use a trivial model of cars to do so. It is illustrated in Figure 3.2. Here, at M0, we have two cars with licence plates "123-ABC" and "456-DEG". At M1, the two cars are abstracted to the

Example

extant) modelling languages (at that time often mis-called methodologies)." These *retro-fitting* steps were key to the modern understanding of the role of metamodels in modeling languages.

5 Seidewitz calls this a *reflexive* metametamodel [Sei03] whereas Álvarez *et al.* [ÁES01] favour the term *meta-circular*, but both are used with equivalent meaning.

6 Jörges *et al.* refer to these meta-layers as *metalevels* [Jör13] (p. 17).

7 The historical context in which the four-layer metamodeling hierarchy emerged is quite interesting and illuminating with regards to its spirit. Henderson-Sellers *et al.* [HS+13] chronicle it quite vividly: "Around 1997, the OMG first publicized their strict metamodelling hierarchy [...] apparently based on theoretical suggestions of Colin Atkinson, not published until a little later [...]. The need for a multiple level hierarchy [...], thus extending the two level type-instance model, was seen as necessary in order to 1) provide a clear means by which elements in the (then emergent) modelling language of UML could be themselves defined *i.e* an M3 level and 2) acknowledge the existence at the M0 level of individual (instances) of the classes designed at the M1 level — although for the OMG/UML world these were seen as less important because such instances only exist as 'data' within the computer program and, in general, do not appear within the modelling process."

class `Car`, with a single attribute of type `String`: `LicencePlate`. At M2, these concepts are further abstracted to the notions of a `Class` and `Attribute`. `Car` is an instance of a `Class`, and its property `LicencePlate` is an instance of `Attribute`. Finally, at M3, we introduce `Class`; M2's `Class` and `Attribute` are both instances of M3's `Class`, as is M3's `Class` itself.

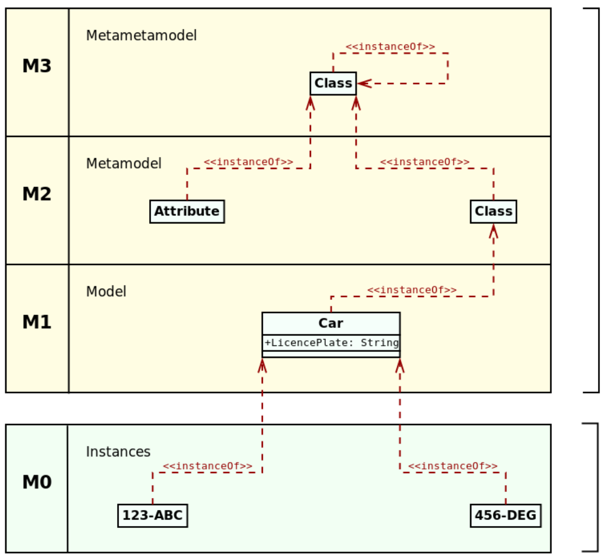


Figure 3.2: Example of the four-layer metamodel architecture. *Source*: Author's drawing based on a diagram by Brambilla *et al.* [BCW12] (p. 16)

Unambiguous terms

The four-layer metamodel architecture has important properties. For example, whilst terms "model" and "meta" are often used in a relative (and even subjective) manner, within the architecture they now become concise — *i.e.*, *metametamodel* is an unambiguous term within this framework. In addition, it was designed as a *strict metamodeling framework*. Atkinson and Kühne explain concisely the intent (*emphasis theirs*):

Strict metamodeling

> Strict metamodeling is based on the tenet that if a model A is an instance of another model B then every element of A is an instance-of some element in B. In other words, it interprets the instance-of relationship at the granularity of individual model elements. The doctrine of strict metamodeling thus holds that the instance-of relationship, and *only* the instance-of relationship, crosses meta-level boundaries, and that every instance-of relationship must cross exactly one meta-level boundary to an immediately adjacent level. [AK02]

Hierarchy limitations

Strict metamodeling is not the only possible approach — Atkinson and Kühne go on to describe *loose metamodeling* on the same paper — nor is the four-layer metamodel hierarchy itself free of criticism. On this regard, we'd like to single out the thorough work done by Henderson-Sellers *et al.* [HS+13], who scoured the literature to identify the main problems with the architecture, and surveyed proposed "fixes", which ranged from small evolutionary changes to "paradigm shifting" modifications. Their work notwithstanding, our opinion is that, though the four-layer metamodel has limitations, it forms a reasonably well-understood abstraction which suffices for the purposes of our own research.

Metamodel diversity

An additional point of interest — and one that perhaps may not be immediately obvious from the above diagrams — is that MDE encourages the creation of "multiple metamodels", each designed for a specific purpose, though ideally all conforming to the same metametamodel. As a result of this metamodel diversity — as well as due to other scenarios described on the next section — operations performed on models have become key to the modeling approach.

3.5 MODELS AND THEIR TRANSFORMATIONS

Source and target models

The second most significant component of MDE, after models, are *Model Transformation (MT)* or just *transforms*. MT are functions defined over metamodels and applied to their instance models. MT receive one or more arguments, called the *source models*, and typically produce one or more models, called the *target models*. Source and target models must be formal models, and they may all conform to the same or to different metamodels.

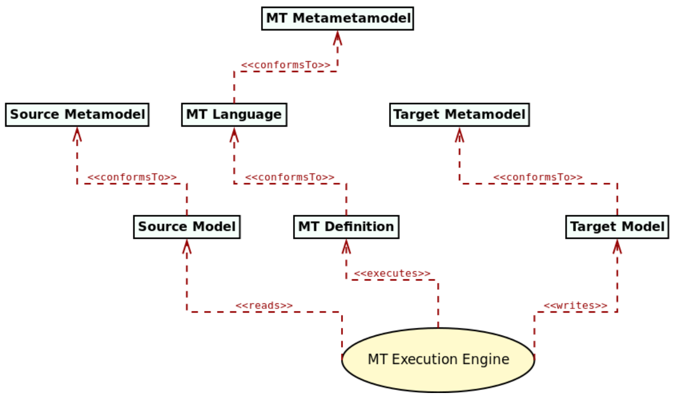


Figure 3.3: Basic model transformation concepts. *Source*: Author's drawing based on diagrams by Brambilla *et al.* [BCW12] (p. 18) and Czarnecki and Helsen [CH06].

MT languages

The literature has long considered MT themselves as models [Béz05b], thus formalisable by a metamodel and giving rise to the notion of *Model Transformation (MT) languages*; that is, modeling languages whose domain is model transformations. MT languages are an important pillar of the MDE vision because they enable the automated translation of models at different levels of abstraction. Figure 3.3 provides an example of how MT languages work, when transforming one type of model to another. However, these are not the only type of MT found in the literature.

3.5.1 *Taxonomy*

The taxonomy of MT has been investigated in great detail in the literature, particularly by Mens and Van Gorp [MVG06] as well as by Czarnecki and

Helsen [CH06]. For the purposes of our dissertation we are primarily concerned with what Czarnecki and Helsen identified as the top-level categories of MT: Model-to-Model (M2M) and Model-to-Text (M2T). These they describe as follows:

M2M, M2T

> The distinction between the two categories is that, while a model-to-model transformation creates its target as an instance of the target metamodel, the target of a model-to-text transformation is just strings. [...] Model-to-text approaches are useful for generating both code and non-code artifacts such as documents.

MT chains

For completeness, there are also Text-to-Text (T2T) transforms, which merely convert one textual representation into another. Transforms of these three types are typically orchestrated into graphs — often called *transform chains* or MT chains[8] — with M2T typically being the ultimate destination. These relationships are illustrated in Figure 3.4, which portraits MT in the wider MDE domain, including models and metamodels.

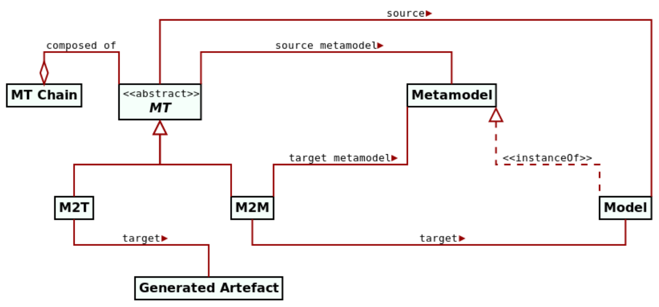


Figure 3.4: Relationships between MT, metamodels and models. *Source:* Author's drawing based on Stahl *et al.*'s diagram [Völ+13] (p. 60).

The importance of MT in MDE stems largely from their broad range of applications — as discussed in the next section.

3.5.2 *Applications*

The use of MT within MDE is pervasive, as demonstrated by Czarnecki and Helsen's non-exhaustive list of intended applications [CH06]:

Intended applications

- **Synchronisation**: The mapping and synchronisation of models, either at the same level of abstraction or at different levels, to ensure that updates are correctly propagated;
- **Querying**: Using queries to generate views over a system;
- **Evolution**: Tasks related to the evolution and management of models such as refactoring, and metamodel updating;

8 Wagelaar's analysis of particular interest in this regard: "Composition of model transformations allows for the creation of smaller, maintainable and reusable model transformation definitions that can scale up to a larger model transformation." [Wag08]

- **Reverse-Engineering**: The generation of high-level models from either source code or lower-level models.
- **Code Generation**: The refinement of high-level models into lower-level models and ultimately to source code — for some, a defining characteristic of the MDE approach.[9]

Given this large number of applications, it is unsurprising that a correspondingly large number of MT languages have emerged over time, including Query / View/ Transformation (QVT) [Kur07], Atlas Transformation Language (ATL) [Jou+08], Epsilon [KPP08], Kermeta [JBF09] and many others. Whilst it is undoubtedly a positive development that many different avenues are being actively explored, there are clearly downsides to this proliferation of solutions: the onus is now on the practitioner to choose the appropriate MT language, and often a deep knowledge of both MDE and the MT languages in question is required to make an informed decision. This apparent *paradox of choice*, at all levels, is one of the biggest challenges faced by MDE, as evidenced by adoption research.

Paradox of choice

A related problem is that, whilst MT languages have many diverse applications, they are ultimately still computer languages and thus prone to suffer from the very same malaises already diagnosed in traditional software engineering. As their use grows, issues such as technical debt [Lan+18], refactoring and difficulties around reuse [Bru+18] will become increasingly pressing. Indeed, these and other similar issues are not specific to MT languages, but shared by *all modeling languages*. It is therefore crucial to understand the *purpose* of modeling languages and clarify their relationship with traditional programming languages.

Technical debt, refactoring, reuse difficulties

3.6 modeling languages and their purposes

The literature commonly distinguishes between two classes of modeling languages, according to their purpose [BCW12] (p. 13):

- **General Purpose Modeling Languages (GPMLs)**: These are languages that are designed to target the modeling activity in the general case, and as such can be used to model *any* problem domain; the domain of these languages is the domain of modeling itself. The UML [OMG17] is one such language.[10]

Modeling language types

- **Domain Specific Languages (DSLs)**: These are languages which are designed for a *specific purpose*, and thus target a well-defined problem domain. They may have broad use or be confined to a small user base such as a company or a single application. As an example, the authors

9 In Jörges words [Jör13] (p. 19): "Code generation is thus an enabling factor for allowing real model-driven software development which treats models as primary development artifacts, as opposed to the approach termed model-based software development [. . .]."

10 A simplification; technically, UML is a *modeling language suite* rather than a modeling language because it is comprised of a number of modeling languages designed to be used together. Note also that not all UML models are formal models but UML models can be made formal through the use of *UML Profiles* and a formal definition of static semantics.

> report in [**marco_craveiro_2021**] on the experiences and challenges of a financial company creating their own modeling DSL.

Figure 3.5 captures how modeling languages relate to the terminology introduced thus far.

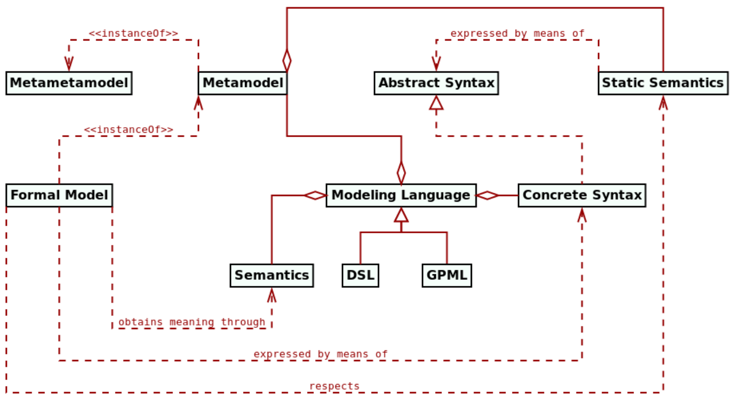


Figure 3.5: Fundamental MDE terminology. *Source*: Author's drawing based on Stahl *et al.*'s image [Völ+13] (p. 56)

Modeling languages as synonyms to DSL

As with most MDE terminology, this classification is not universally accepted. For instance, Stahl *et al.* [Völ+13] (p.58) take the view that (*emphasis theirs*) "[often] the term *modeling language* is used synonymously with DSL. We prefer the term DSL because it emphasizes that we always operate within the context of a specific domain." Somewhat unexpectedly, Jörges *et al.* [Jör13] (p. 15) agree with this stance. We are instead of the opinion that conflating DSL with modeling languages hinders precision unnecessarily and thus is not a useful development. For the remainder of this work, we shall use the three terms (modeling language, General Purpose Modeling Language (GPML) and DSL) with specifically the above meanings.[11] The crux of the problem is, then, in deciding *how* a given modeling problem is to be tackled.

3.7 DETERMINING THE MODELING APPROACH

One of the first decisions faced by practitioners when when modeling a problem is the choice between using GPMLs or DSLs. Mohagheghi and Aagedal's analysis highlights the kinds of trade-offs that must be considered [MA07]:

Trade-offs

> A metamodel's conceptual complexity should lead to greater expressive power, and thus smaller models in size. For example, modeling languages developed for a specific domain [e.g. DSLs] have more expressive power and are closer to the experts' knowl-

11 In Stahl *et al.*'s defence, most GPML require a degree of extensibility in order to support formal models — *e.g.*, in the case of UML, the creation of profiles are typically required, and thus considered a DSL. In this sense, *plain* GPMLs appear to be of a limited use in a MDE context. Nonetheless, our point is that, even without further customisation, GPMLs provide a sufficient basis for simpler automation use cases. Therefore, in our opinion, it is incorrect to think of GPMLs as "merely" tools for some form of Model Based Engineering (MBE).

edge of the domain than general-purpose modeling languages [GPMLs], but may be more complex to learn for a novice.

DSL creation

If opting for DSLs, their creation can be achieved either via extensibility mechanisms available in most GPMLs — such as the before-mentioned UML profiles — or by in-house language design. Regardless of the approach, the use of DSLs is strongly encouraged within MDE since, as Stahl *et al.* tell us (p. 58), "[a] DSL serves the purpose of making the key aspects of a domain — but not all of its conceivable contents — formally expressable and modelable." [Völ+13]

XMLware, JSONWare

A vital component of language design is choosing a notation and implementing its parsing. A simple alternative — ideal for basic requirements —is to use an existing markup language such as XML or JSON, in order to take advantage of their strong tooling ecosystem. In the XML case, an XML schema can be defined using XSD, so as to constrain XMLs concrete syntax. The abstract syntax will be dependent on how the XML processing is performed — *i.e.* using Document Object Model (DOM), Simple API for XML (SAX) or any other XML Application Programming Interface (API). The approach is commonly referred to as *XMLware* [Arc+11], but is not without its detractors [Neu16].[12]

Grammarware

Yet another alternative, attractive for text-based notations, is to define a formal grammar for the concrete syntax (*cf.* Section 3.2.1) using a parser generating tool such as YACC [Joh+75] or Bison [DS92], and then code-generate a parser for the grammar in a general purpose programming language. The parser is responsible for processing documents written in the concrete syntax and, if valid, instantiating an abstract syntax tree: a tree representation of the abstract syntax. This approach is called *grammarware* in the literature [KLV05; PKP12], and has historically been used to define programming languages, but it is equally valid for modeling textual DSLs.[13]

Modelware

Finally, the more modern take on this problem is called *modelware* because it relies on model-driven principles and tooling. It is implemented using tools such as XText [EB10], which generate not just the parser but also customisable abstract syntax, as well as providing IDE support for the newly-designed language. Modelware is the preferred approach within the MDE community because it embodies many of the core principles described in this chapter, and, in addition, some modelware stacks offer integrated support for graphical notations.[14]

Nonetheless, regardless of the specifics of the approach, there are clear similarities between traditional programming languages and modeling languages, as we shall see next.

12 An XMLWare-based stack was the solution used by a financial company, whose experiences are narrated in [**marco_craveiro_2021**].

13 Several programming languages also offer modern parser libraries, such as C++'s Boost Spirit [Boo18] or Java's ANTLR [Par13]. These are more appealing to typical software developers (as opposed to compiler writers) because they provide a better fit to their workflows.

14 Seehusen and Stølen 's evaluation report of Graphical Modeling Framework (GMF) serves as a typical example [SS11]. GMF is part of the vast Eclipse Modeling Framework (EMF) modelware stack; those specifically interested in EMF are directed instead to Steinberg *et al.* [Ste+09].

3.8 MODELING LANGUAGES AND PROGRAMMING LANGUAGES

There are obvious advantages in clarifying the relationship between programming languages and modeling languages, because the latter can benefit from the long experience of the former. Predictably, the literature has ample material on this regard. Whilst discoursing on the unification power of models, Bézivin spoke of "programs as models" [Béz05b]; France and Rumpe tell us that "[source] code can be considered to be a model of how a system will behave when executed." [FR07] Indeed, from all that has been stated thus far, it follows that all general purpose programming languages such as C++ and Java can rightfully be considered modeling languages too and their programs can be thought of as models implemented atop a grammarware stack.

Source code as a model

Here we are rescued by Stahl *et al.*, who help preserve the distinction between programming languages and modeling languages by reminding us that they have different responsibilities (*emphasis ours*): "The means of expression used by models is geared toward *the respective domain's problem space*, thus enabling abstraction from the programming language level and allowing the corresponding compactness." [Völ+13] (p. 15) That is, programming languages are abstractions of the machine whereas modeling languages are abstractions of higher-level problem domains — a very useful and concise separation.[15] With this, we arrive at the taxonomy proposed by Figure 3.6.

Languages and abstraction

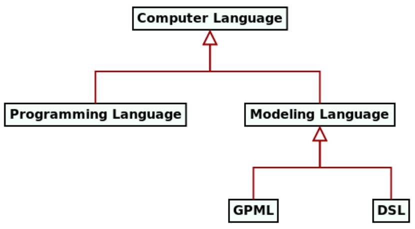


Figure 3.6: Taxonomy of computer languages within MDE.

A related viewpoint from which to look at the relationship between modeling languages and programming languages is on how information can be propagated between the two via MTs (*cf.* Section 3.5). The simplest form is via *forward engineering*, whereby a model in a modeling language is transformed into a programming language representation, but changes made at the programming language level are not propagated back to the modeling language.

Forward engineering

The converse happens when using *reverse engineering*: a model in a modeling language is generated by analysing and transforming source code in a programming language. Clearly, there are difficulties in any such an endeavour due to the mismatch in abstraction levels, as explained above. Finally, the

Reverse and roundtrip engineering

15 These camps are not quite as distinct as they may appear, and many are working to shorten the differences. On one hand, there are those like Madsen and Møller-Pedersen who propose a more direct integration of modeling concepts with programming languages themselves [MMP10]. On the other hand, there are also those like Badreddin and Lethbridge, proponents of Model Oriented Programming (MOP), who defend making modeling languages more like programming languages [BL13]. Both approaches show a great deal of promise but, given their limited application at present, we declined to investigate them further.

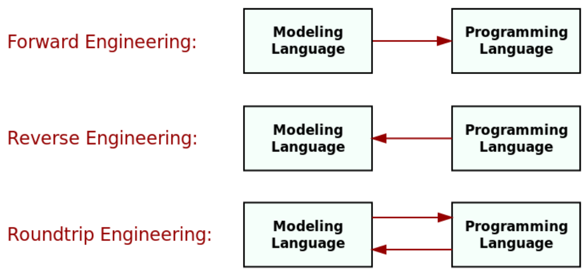


Figure 3.7: Propagating information between modeling languages and programming languages. *Source*: Author's drawing based on an image from Stahl *et al.* [Völ+13] (p. 74).

most difficult of all scenarios is Round-Trip Engineering (RTE), in which both modeling and programming language representations are continually kept synchronised, and changes are possible in either direction. Figure 3.7 illustrates these three concepts.

As briefly alluded to in Section 3.5, these and other related topics fall under the umbrella of *model synchronisation* within MDE literature. They are addressed in the next Chapter (*cf.* Chapter 4), which starts to delve in more detail into MDE's aspirations of matching modeling languages to different *abstraction levels*.

4 FROM PROBLEM SPACE TO SOLUTION SPACE

> *In particular, current research in the area of model driven engineering (MDE) is primarily concerned with reducing the gap between problem and software implementation domains through the use of technologies that support systematic transformation of problem-level abstractions to software implementations.*
>
> — France and Rumpe [FR07]

Software engineering typically distinguishes between problem space (or problem domain) and solution space.[1] The present chapter's objective is to clarify these two very important concepts, and to relate them to the practice of model-driven approaches.

Chapter overview

The chapter is organised as follows. Section 4.1 connects *abstraction levels* to these two different spaces. The chapter finishes by interrogating in more detail the composition of the solution space (Section 4.2), with a particular emphasis on the concepts of *platforms* (Section 4.2.1) and *Technical Spaces (TSs)* (Section 4.2.2).

4.1 SPACES AND LEVELS OF ABSTRACTION

Problem space, Solution space

The *problem space* concerns itself with a business area or any other field of expertise a software system needs to be developed for, whereas the *solution space* is made up of a set of technological choices with which a software system can be designed and implemented, and atop of which it will execute.[2] As we move from problem space towards solution space, the abstraction level is progressively lowered until the machine is ultimately reached. Figure 4.1 illustrates this idea.

One of the core objectives of MDE is to enable a smoother transition between abstraction levels, easing the gap between them. France and Rumpe lay out the motivation (*emphasis ours*):

Problem-implementation gap

> A *problem-implementation* gap exists when a developer implements software solutions to problems using abstractions that are at a *lower level* than those used to express the problem. In the

1 For a treatment of the subject in a system's engineering context, see Chapter 14 of Wasson [Was15] (p. 135).

2 In the words of Groher and Völter's: "The problem space is concerned with end-user understandable concepts representing the business domain of the product line. The solution space deals with the elements necessary for implementing the solution, typically IT relevant artifacts." [GV09].

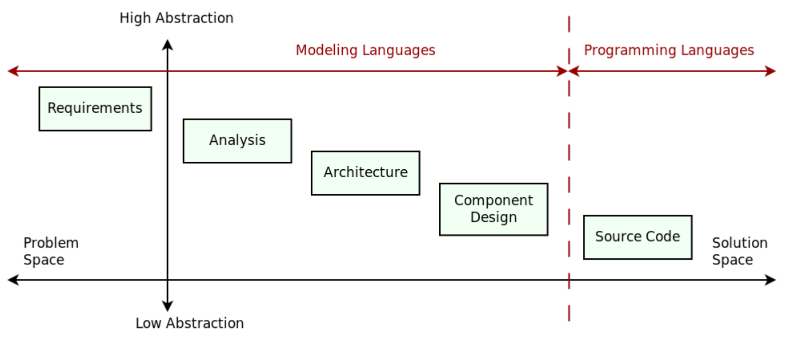


Figure 4.1: Phases of the software development lifecycle, abstraction levels and language types. *Source*: Author's drawing based on Berg *et al.*'s [BBM05] image.

> case of complex problems, bridging the gap using methods that rely almost exclusively on human effort will *introduce significant accidental complexities* [FR07].[3]

Cascading, Refinement

Hence, MDE promotes the use of modeling languages (*cf.* Section 3.6) at the appropriate level of abstraction for the task at hand; its ultimate goal is to allow software engineers to create a cascading set of abstractions of an arbitrary depth that closely matches software engineering activities over the various phases of the software development lifecycle, with each abstraction described by an adequate modeling language — all the way to the general purpose programming language. This process is referred to as *model cascading*, and it is implemented by means of *model refinement* (*cf.* Section 3.5.2).

Synchronisation and integration challenges

Whilst conceptually straightforward, model cascading poses awkward practical challenges because the existence of multiple models, possibly conforming to multiple metamodels, and representing disparate viewpoints leads to a need to keep all views integrated, synchronised and consistent[4] — a task of increasing difficulty, as MDE moves away from the simpler unidirectional model of transformations towards more complex topologies.[5] As we've already seen (*cf.* Section 3.6), similar synchronisation and integration challenges are also present in the relationship between models and source code — the traditional destination of the model refining process.[6]

3 Whittle *et al.* define *accidental complexity* as (*emphasis ours*): "[. . .] where the tools introduce complexity *unnecessarily* [Whi+17]." What is meant by *unnecessarily*, of course, is left as an exercise to the reader.

4 Much has been written in the MDE literature about model synchronisation and integration, but it lays beyond the scope of the present study. The interested reader is directed to Giese *et al.* [GHN10] for an introductory overview of model integration and model synchronisation (Section 2, State of the Art), and to Hettel *et al.* [HLR08] for an analysis of RTE in the context of MDA, but largely applicable to MDE in general. Czarnecki and Helsen's MT Feature Model is also relevant (Section "Source-Target Relationship" [CH06]

5 Diskin *et al.* see beyond simple cascading and speak instead of networks of models (*emphasis ours*): "A pipeline of unidirectional model transformations is a well-understood architecture for model driven engineering tasks such as model compilation or view extraction. However, modern applications require a shift towards *networks of models related in various ways*, whose synchronization often needs to be incremental and bidirectional." [Dis+14]

6 The alternative to code generation is model-based execution, either via an interpreter or compilation. It is, however, outside the remit of the present work. For a treatment of the subject in the context of UML, see Mellor *et al.* [MBFBJ02]

Difficulties in synchronising source code with models can be avoided if *full code generation* is targeted, a task considered feasible by some — such as Jörges *et al.* [Jör13] (p. 33) — and unfeasible by others, such as Greifenberg *et al.* [Gre+15b], who state: "The prevailing conjecture, however, is that deriving a non-trivial, complete implementation from models alone is not feasible." From experience, we lean more towards Greifenberg *et al.* in this regard. The alternative is to use *partial code generation*, but then there is a requirement for one or more *integration strategies* to allow handcrafted and generated code to coexist. Here, Greifenberg *et al.*'s survey of integration mechanisms is extremely helpful [Gre+15b; Gre+15a].

Full and partial code generation

In our personal opinion, largely borne out of practical experience, model synchronisation remains a complex subject with thorny problems — both engineering and theory-wise — and one which is particularly difficult to address at a large, industrial scale. For these reasons, the present work recommends relying mainly on the simpler forward-only topology, with minimal use of cascading, and resorting to well defined integration strategies; and to adopt more complex approaches *solely* when well-defined use cases emerge.

Recommended approach

Complexity notwithstanding, we have thus far only scratched the surface of the solution space. The next section identifies its key components and their properties.

4.2 THE STRUCTURE OF THE SOLUTION SPACE

There are a few nuances to add to the simplified picture described in the previous section because the underlying process is of a fractal nature.[7] That is, by looking in more detail at each step on our abstraction descend, we will likely find inside it yet another abstraction ladder. Consider the solution space. Within it, the literature typically defines two key concepts: *Technical Space (TS)* and *Platforms*. Figure 4.2 illustrates how they relate to each other and to the problem and solution spaces. These two concepts are of vital importance to us, so the next two sections will analyse them in detail, including a discussion of the challenges they present.

Solution space components

4.2.1 *Technical Spaces*

Kurtev *et al.* proposed TSs in their seminal paper [KBA02], defining them as follows: "A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities." Mens and Van Gorp subsequently updated the language and tightened the notion by connecting it to metametamodels: "A technical space is determined by the metametamodel that is used (M3-level)." [MVG06] (*cf*, Section 3.4)

Definition

7 This is to be expected, given that abstractions can be composed of other abstractions by means of Stachowiak's mapping feature (*cf.* Section 3.1). In particular (*emphasis his*): "[models] are models of *something*, namely, [they are] reflections, representations of natural and artificial originals, that can themselves be models again." [Sta73]

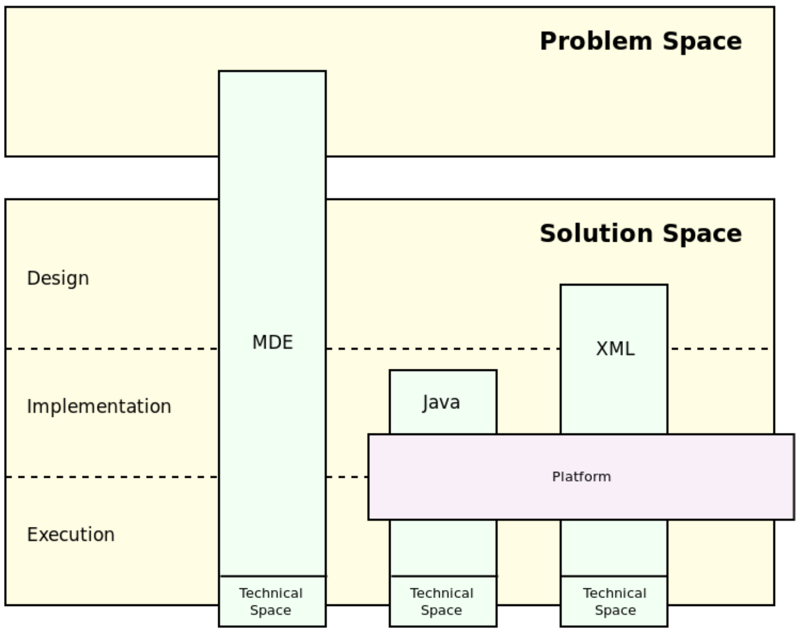


Figure 4.2: Problem space, solution space, TSs and platforms. Author's drawing based on Brambilla *et al.*'s image [BCW12] (p. 13)

Examples of TSs include MDE itself, XML, Java and other such programming languages.

Motivation

In [Béz+03], Bézivin *et al.* outlines their motivation: "The notion of TS allows us to deal more efficiently with the ever-increasing complexity of evolving technologies. There is no uniformly superior technology and each one has its strong and weak points." The idea is then to engineer bridges between technical spaces, allowing the importing and exporting of artefacts across them. These bridges take the form of adaptors called "projectors", as Bézivin explains (*emphasis ours*):

Projectors, injectors, extractors

> *The responsibility to build projectors lies in one space*. The rationale to define them is quite simple: when one facility is available in another space and that building it in a given space is economically too costly, then the decision may be taken to build a projector in that given space. There are two kinds of projectors according to the direction: *injectors* and *extractors*. Very often we need a couple of injector/extractor to solve a given problem. [Béz05a]

Challenges

TSs are a useful — if somewhat imprecise[8] — conceptual device and bridging across them has been demonstrated to work in practice [Béz+03]. However, our position is that to fully fulfil their promise, an extraordinary engineering effort is required to model all significant features from existing TSs, to expose

8 In the words of Bézivin *et al.* [Béz+03] (*emphasis ours*): "Although *it is difficult to give a precise definition of a Technological Space*, some of them can be easily identified, for example: programming languages concrete and abstract syntax (Syntax TS), Ontology engineering (Ontology TS), XML-based languages and tools (XML TS), Data Base Management Systems (DBMS TS), Model-Driven Architecture (MDA TS) as defined by the OMG as a replacement of the previous Object Management Architecture (OMA) framework."

them to modeling languages and to keep those models updated. As we shall see in the next section, much of the same challenges apply to platforms.

4.2.2 *Platforms*

Colloquial term

The term *platform* is employed within the software engineering profession in a broad a variety of contexts, from hardware to operative systems, compilers, IDEs like the Eclipse Platform[9], virtual machines providing programming environments such as the Java Virtual Machine (JVM) and the Common Language Runtime (CLR), and in numerous other cases. It is also a core term within MDE, and a foundation upon which many other concepts build, so it is important to arrive at a clear understanding of its meaning.

The literature often uses the MDA definition as a starting point, stated as follows:

MDA Definition

> A platform is the set of resources on which a system is realized. This set of resources is used to implement or support the system. In the context of a technology implementation, the platform supports the execution of the application. Together the application and the platform constitute the system. [Gro14] (p. 9)

Versus MDE

From a software engineering standpoint, a platform is often seen as mechanism for reuse and abstraction, but MDE goes further and considers as particularly useful those that are "semantically rich" and "domain-specific", made up of "prefabricated, reusable components and frameworks [because they] offer a much more powerful basis than a ’naked’ programming language or a technical platform like Java Platform Enterprise Edition (J2EE)." [Völ+13] (p. 15)

Versus TS

Figures 4.2 and 4.3 explore these ideas by depicting the relationship between platforms and TSs. From this perspective, TSs provide the raw building materials and platform developers leverage their technical expertise to, in the words of Brambilla *et al.*, "combine them into a coherent platform" [BCW12] (p. 13). By sitting atop a platform, software engineers can abstract themselves from lower-level implementation details and focus on the problem at hand.

Versus problem-implementation gap

In the presence of code generation, a tempting alternative may appear to be to bind the building blocks directly against a modeling language. Experience has however demonstrated the pitfalls of this approach, and here we are once more faced with the familiar theme of a need to *raise the abstraction level*. In practice, the building blocks found in TSs are at too low a level to make them suitable for direct integration with a modeling approach because, as already discussed (*cf.* Section 4.2), bridging the abstraction gap becomes *increasingly difficult as the gap widens*. Stahl *et al.* agree, but focus instead on the converse, stating that "[the] platform has the task of supporting the realization of the domain, that is, the transformation of formal models should be as simple as possible. [...] Clearly, the easier the transformations are to build, the more powerful is the platform." [Völ+13] (p. 61) France and Rumpe follow

9 https://projects.eclipse.org/projects/eclipse.platform

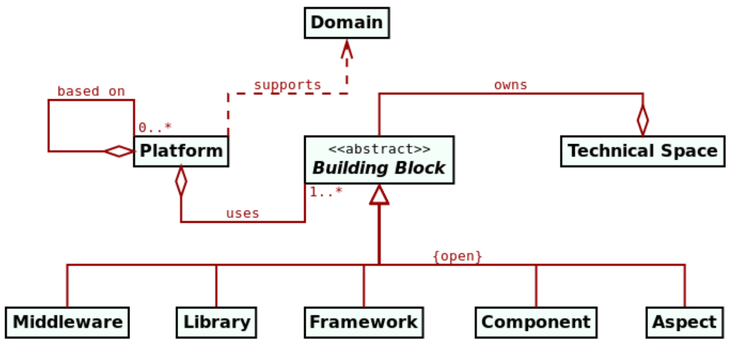


Figure 4.3: Platforms and associated concepts. *Source*: Author's drawing based on Stahl *et al.*'s image [Völ+13] (p. 59)

the same line of reasoning, positing that abstractions such as platforms are key, because "[the] introduction of technologies that effectively raise the implementation abstraction level can significantly improve productivity and quality with respect to the types of software targeted by the technologies." [FR07]

Challenges posed by constant evolution

Unfortunately, not all is positive. On the same paper, France and Rumpe leave a decidedly stark warning about the challenges created by the very same process: "[the] growing complexity of newer generations of software systems can eventually overwhelm the available implementation abstractions, resulting in a widening of the problem-implementation gap." In other words, modeling languages close to a platform can only remain relevant if they are continually kept up to date with the constant changes to the platforms they depend on, or else risk becoming obsolete. This is a very difficult problem to tackle.

PIM, PSM

An obvious way to mitigate issues that arise from the constant platform churn is to decouple platform-dependent concepts from those that are independent of a target platform. This partitioning — originally popularised within MDA but now rightfully considered a part of mainstream MDE — does not directly address the underlying causes but does have the advantage of reducing the overall impact surface. As a result, by classifying models with regards to their dependence on a platform, we arrive at the notion of Platform Independent Models (PIMs) and Platform Specific Models (PSMs). In [Völ+13] (p. 20), Stahl *et al.* explain that "[...] concepts are more stable than technologies [...]. The PIM abstracts from technological details, whereas the PSM uses the concepts of a platform to describe a system." A secondary advantage of this approach is that a single PIMs can be mapped to multiple PSMs, as demonstrated by Figure 4.4.

However, when one looks at these elegant solutions in more detail, the literature enters once more difficult terrain. First and most significantly, there are still looming challenges in establishing just what exactly a platform *is*. Bézivin explains the matter rather eloquently (*emphasis ours*):

> There is a considerable work to be done to characterize a platform. How is this related to a virtual machine (*e.g.* JVM) or to a specific language (*e.g.* Java)? How is this related to a general implemen-

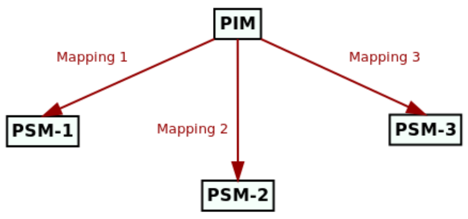


Figure 4.4: Mapping between a PIM and three PSM. *Source*: Author's drawing based on Stahl *et al.*'s image [Völ+13] (p. 20)

> tation framework (*e.g.* DotNet or EJB) or even to a class library? How to capture *the notion of abstraction between two platforms*, one built on top of the other one? *The notion of a platform is relative* because, for example, to the platform builder, the platform will look like a business model. One may also consider that *there are different degrees of platform independence*, but here again no precise characterization of this may be seriously established before we have an initial definition of the concept of platform. [Bézo5b]

Characterisation challenges

Secondly, there is the question of how the mappings are to be achieved. In the same paper, Bézivin suggested employing Platform Description Models (PDMs) as a way to bridge this gap — that is, the use of models to describe the capabilities of platforms. This and several other ideas informed research, which became very active and produced a number of localised solutions, for example in the context of MDA [WJ05] and XML [Neu16]. Nonetheless, a general approach to the problem remains illusive, as Anjorin *et al.* explain (*emphasis theirs*):

Transformation and mapping challenges

> Although there exist numerous strategies and mature tools for certain isolated subtasks or specific applications, a general *framework* for designing and structuring model-to-platform transformations, which consolidates different technologies in a flexible manner, is still missing, especially when *bidirectionality* is a requirement. [Anj+12]

Lack of a framework

Their work provides an informed summary of the state of the art on this regard, as well as proposing a promising direction for such a generalised framework; nevertheless, substantial research and engineering work remains, in order to address all of the issues highlighted above.

Thirdly, there are those who question the need to make PSMs explicitly visible, asking whether they are not best seen as a conceptual device and a (hidden) implementation detail. Stahl *et al.* report that "[p]ractical project experience has hitherto proved that this simplification [of foregoing explicitly visible PSMs] is usually more useful than the additional degrees of freedom gained with PSMs." [Völ+13] (p. 24) According to them, such a simplification permits more efficient development and reduces the thorny issues around model synchronisation, particularly from lower to higher levels of abstraction — *i.e.*, the propagation of changes from PSMs to PIMs (*cf.* Section 4.2).[10]

Hidden PSM

10 These are, in effect, merely a variation of the RTE problem described in Section 3.8.

Platform uncertainties

In light of all of these difficulties, and even taking into account the Pragmatism Principle, one is nevertheless forced to conclude that Bézivin's words of warning still to loom large over the field: "Answering the question of what is a platform may be difficult, but until a precise answer is given to this question, the notion of platform dependence and independence (PSMs and PIMs) may stand more in the marketing than in the technical and scientific vocabulary." [Béz05b]

These stimulating words complete our sketch of the solution space and its challenges. Our attention shall now turn "upwards" once more, towards the bigger picture, as we investigate the interaction between MDE and the various methodologies and processes used for the development of software systems.

Part II

INTEGRATIONS

5 MDE AND THE SOFTWARE DEVELOPMENT PROCESS

One of the more controversial concepts in Agile Modeling is that agile models and agile documents are sufficient for the task at hand, or as I like to say they are "just barely good enough". For some reason people think that just barely good enough implies that the artifact isn't very good, when in fact nothing could be further from the truth. When you stop and think about it, if an artifact is just barely good enough then by definition it is at the most effective point that it could possibly be at.

— Ambler [Amb07]

Motivation

Model Driven Engineering is deeply intertwined with the software development lifecycle; and the shape of the software development lifecycle is a function of the prevailing Software Development Methodology (SDM).[1] The present chapter provides and overview of this complex relationship, pointing out implications of this interleaving. SDMs and related processes have themselves been treated extensively in the software engineering literature, including numerous comparative studies and surveys [Des14; Awa05; SVI12; DBC88; Est+07]. In the interest of maintaining focus, the objective of the present section is not to perform an exhaustive review across this vast field but to: a) supply a set of conceptual building blocks to be used within this dissertation; and b) to investigate the relationship between MDE and SDMs, including a characterisation of the general challenges posed by MDE to the development process and vice-versa.

Chapter overview

The chapter is organised as follows. It begins by defining more precisely what is meant by an SDM (Section 5.1). We then attempt to determine if MDE has a "preference" towards certain types of SDM's on Section 5.2. The chapter concludes by looking at *two-track development* (Section 5.3), an approach commonly used in the presence of MDE.

1 A distinction has been preserved in this dissertation between the "software development lifecycle" and "Software Development Lifecycle (SDLC)". The Software Development Lifecycle (SDLC) is considered a SDM by many authors such as Elliot [Ell04] (p. 86-87), who calls it the "oldest formalised methodology for building information systems". He states (*emphasis his*):

> The traditional approach to information systems development was known as the *waterfall* approach or *systems development life cycle* approach — the SDLC approach. [. . .] This methodology pursues the development of information systems in a very deliberate, structured and methodical way, requiring each stage of the life cycle, from inception of the idea to the delivery of the final system, to be carried out rigidly and sequentially."

Conversely, this document employs "software development lifecycle" in a more informal manner, meaning *any* lifecycle model — often a property of the underlying SDM. For those interested in a more rigorous approach, Estefan *et al.* [Est+07] provide a detailed discussion on the subject, defining these and other related terms. In particular, see Section 2 — "Differentiating Methodologies from Processes, Methods, and Lifecycle Models".

5.1 SOFTWARE DEVELOPMENT METHODOLOGIES

Definition

SDMs have had a important role to play in the development of software systems from as early as the 1960's [Ell04] (p. 86). According to Ramsin and Paige, "[a] SDM is a framework for applying software engineering practices with the specific aim of providing the necessary means for developing software-intensive systems." [RP08] For their part, Avison and Fitzgerald see it as "a recommended collection of phases, procedures, rules, techniques, tools, documentation, management, and training used to develop a system." [AF03] Based on their analysis of earlier work, including that of OMG [OMG17], as well as Avison and Fitzgerald themselves [AF03], Ramsin and Paige decomposed SDMs into three main parts:

Components

- **Philosophy**: a set of assumptions and beliefs made by the authors of the methodology, as part of its motivation or with the intent of supporting and rationalising it.
- **Modeling conventions**: a set of modeling conventions designed to work together as a modeling language. Within this dissertation, we choose to interpret *modeling language* in the sense defined by Chapter 3 — though perhaps Ramsin and Paige's intention is to use the term in a less formal manner.
- **Processes**: a set of processes that provide guidance as to the activities to take place, their order, the role played by different actors in those activities and ways in which to monitor and evaluate the results of performing them.

Component interactions

Within this framework, the modeling language contains the vocabulary with which to model aspects of the development of the system, the processes cater for the temporal characteristics of how entities interact — *i.e. who* performs what activities *when* — and the philosophy supplies a overarching narrative that harmonises and justifies both. These three components come together as a *methodology*, providing a unified perspective from which to accomplish prescribed objectives.[2] Therefore, an SDM can be thought of as an *abstraction* that orchestrates activities related to the development of software systems; and does so with the objective of making them easier to understand for all involved.

It is difficult to overestimate the importance of SDMs in modern software development. Ramsin and Paige [RP08] go as far as considering them an enabler of the software engineering discipline *itself* (*emphasis ours*):

Relationship with software engineering

> Software development methodologies are therefore considered an integral part of the software engineering discipline, since methodologies provide *the means for timely and orderly execution* of the various finer-grained techniques and methods of software engineering.

2 There are those like as Brinkkemper [Bri96] who vehemently disagree with how software engineers employ the terms *method* and *methodology*. Though the points presented by Brinkkemper and others are valid, we have decided to remain aligned with the traditional software engineering usage, given that the meaning is now widely understood amongst practitioners. See also Estefan *et al.* [Est+07] for a thorough discussion of these and related terms.

Whilst SDM's exact importance may be argued, what should nonetheless be clear from this exposition is that their organisational properties are simultaneously orthogonal and complementary to the characteristics of MDE described thus far. It is for this reason that we sided earlier on with those that argue that MDE *by itself* cannot be considered a methodology, but methodologies can be developed with model-driven characteristics for particular purposes. Asadi and Ramsin [AR08] tackled this subject in an MDA context, but presented an argument which, in our opinion, is directly applicable to MDE (*emphasis ours*):

MDE is not an SDM

> [. . .] *MDA is not a methodology*, but rather an approach to software development. This fact forces organizations willing to adopt the MDA to either transform their software development methodologies into Model-Driven Development (MDD) methodologies, or use new methodologies that utilize MDA principles and tools towards the realization of MDA standards.

It is for this reason that one finds numerous examples of model-driven methodologies in the literature, with differing characteristics [Mat11; Amb08; Amb07; GSD09; Est+07]. However, rather than delve into the specifics of each of these, the more pertinent question — at least for the purposes of this dissertation — is to determine whether MDE *demands* a particular type of software development methodology, or has no sensitivity to it.

5.2 ITERATIVE OR STRUCTURED?

The literature reveals a range of views with regards to the relationship between MDE and SDMs. Ambler appears to suggest MDE is more amenable to a structured environment, declaring that (*emphasis ours*) "MDD is an approach to software development where extensive models are created *before* source code is written. With traditional MDD a serial approach to development is often taken where comprehensive models are created early in the lifecycle." [Amb08] Stahl *et al.* are instead of the view that MDE requires, by its very nature, an iterative process:

Structured versus iterative

> The iterative, dual-track process of MDSD [. . .], in which the infrastructure is developed in parallel to the application(s), must be clearly distinguished from traditional waterfall methods that are based on a 'big design up-front' philosophy. [Völ+13] (p. 375)

Unfortunately, the disagreement may stem, at least in part, from imprecise definitions rather than due to profound ideological differences.

A third view, and one which we align ourselves with, is that of Brambilia *et al.*, who declare MDE to be *process-agnostic*, claiming "it neither provides nor enforces any specific development process but it can be integrated in any of them." [BCW12] (p. 53) To be clear, Brambilia *et al.* are not suggesting that MDE does not have a significant impact on a project's software development methodology and vice-versa. Instead, their argument is that the principles in the MDE body of knowledge are compatible with *all* SDMs — "traditional

MDE is agnostic to process

development processes" in their parlance — and the onus is therefore on the MDE practitioner to unify them into a whole, for any particular application.

These integration efforts are not insignificant, so the literature has been active in developing specific strategies — especially for iterative methodologies [Mat11; Amb08; Amb07; GSD09]. It is also important to understand that this stance does not have any implications with regards to the merits (or demerits) of structured versus iterative development methodologies, as these can be analysed independently of MDE. However, since models "are considered equal to code" [Völ+13] and, somewhat more arguably, since "programming is a modeling activity" [MMP10], it is not surprising that many of the engineering practices that foster the development of high-quality code are equally desirable when the engineering is driven by modeling.

Modeling versus coding

It is then for these reasons that iterative methodologies should be preferred to structured methodologies, rather than due to any intrinsic property or requirement of MDE. The idea can be neatly summarised with the following dictum: "what is good for code is (generally) also good for models". Nonetheless, regardless of whether the approach is iterative or structured, there are specific factors related to model-driven software engineering that must be catered for, as the next section will explain.

5.3 TWO-TRACK DEVELOPMENT

One of the most striking differences between traditional software development and the model-driven approach is the additional work required to develop the modeling infrastructure — that which Stahl *et al.* call the *domain architecture* [Völ+13] (p. 253); *i.e.* the set of modeling languages and their associated MTs, as well as the platforms upon which they depend on, in order to translate a set of instance models into a software system. The development of the domain architecture poses a challenge which straddles theory and application because there is a circular dependency between exploring problem and solution spaces *and* creating the vocabulary with which to perform that exploration.

Domain architecture

Stahl *et al.* tackled the issue by proposing a two-track development process, composed of the following threads:[3,4]

Development threads

- **Domain Architecture Development Thread**: Responsible for developing all of the modeling infrastructure that makes up the domain architecture. Abstracts and generalises the requirements produced by the application thread into infrastructure that can be reused for a number of similar applications.

3 Interestingly, Stahl *et al.* first introduce the approach in the context of Architecture-Centric MDSD (AC-MDSD) [Völ+13] (p. 21) but later on generalise it to make it widely applicable to Model Driven Software Development (MDSD) [Völ+13] (Chapter 13, p. 253).

4 For simplicity, we are not making the customary Domain Engineering separation between analysis and development (*cf.* Section 6.1). We do so partially because we take the iterative approach — fusing development and analysis together — but also because we believe the same argument applies to analysis and development.

- **Application Development Thread**: Concerned with producing a concrete product to satisfy the requirements of end users. Provides exemplars of needed functionality to the domain architecture development thread to help shape its direction and consumes the tooling it produces to implement the product.

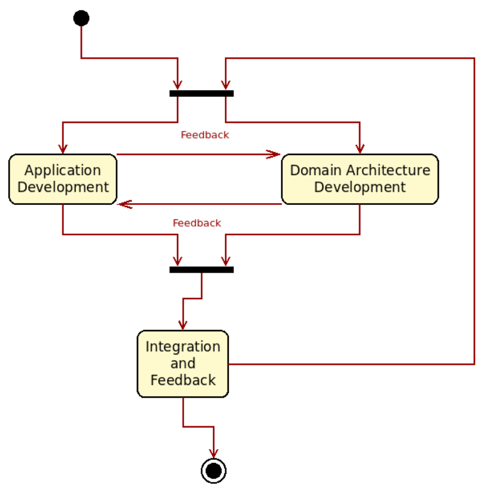


Figure 5.1: Two-track development. *Source*: Author's drawing based on Stahl *et al.*'s image [Völ+13] (p. 262).

The relationship between these two threads is illustrated in Figure 5.1. Two-track development is very useful as an idealised model because the approach evokes a clear separation of concerns. Nonetheless, our own personal experience has taught us that matters are not quite as clear cut in practice. On one hand, the basic principles are very easy to convey to experienced engineering teams — even where MDE knowledge is lacking. On the other hand, many dangers lie in wait of a naïve application:

Challenges

- the before-mentioned circular nature of the problem is extremely challenging — and never more so than during the initial stages of development;
- the development of the domain architecture ultimately demands a *multidisciplinary skill-set*, entailing both good software engineering skills as well as a mastery of the MDE cannon;
- there is great difficulty in demonstrating to management the need for continued investment in domain architecture development and maintenance as the project reaches maturity, and the lack of investment has severe consequences for the long term;
- there is a great difficulty in avoiding a disconnect between domain and architecture development teams, and there are far-reaching repercussions when such a disconnect occurs.

The last factor is of great importance because it may lead to a phenomena we named *problem domain decoupling*, and which happens as the disconnect grows in size. In our opinion, this problem manifests itself more evidently with the application of AC-MDSD, but we do not believe it is solely limited

Domain decoupling

to this use case. Figure 5.2 illustrates the issue by looking at four hypothetical *scenarios* covering the application of MDE to infrastructural code, which we shall now enunciate.

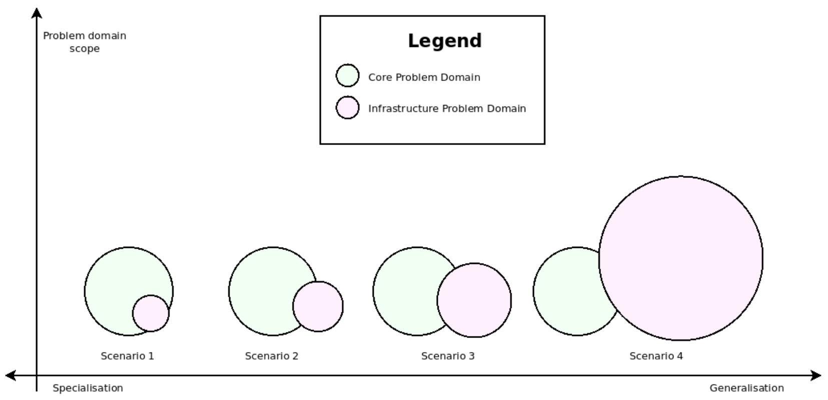


Figure 5.2: Problem domain decoupling.

Scenario analysis

Scenario 1 typically happens at the first brush with MDE, where developers create a solution hard-wired to the core problem domain they are exploring, and is closely related to our own personal experiences [**marco_craveiro_2021**]. From Scenario 2 to Scenario 4 there is a quest for *generalisation*, accompanied by a corresponding *growth in scope* of the infrastructure problem domain. Scenario 2 represents a small decoupling of the infrastructure domain to make it useful to more than one product, though still fairly hard-wired. With Scenario 3 we are now looking at providing infrastructure for a larger grouping of software products and their diverse needs, with a resulting ballooning in infrastructural scope. Finally, with Scenario 4, the infrastructure domain becomes a product on its own right, much larger than any one core problem domain; at this stage we are now considering products supplied by external vendors rather than in-house development.

Scenario 1 versus Scenario 3

The *quantitative change* in the size of the infrastructure problem domain produces *qualitative changes* that may not be readily apparent to engineers, as they develop a system with a dual-track approach. This *phase transition* is particularly problematic as one transitions from Scenario 1 through to Scenario 3. Let us perform a comparison between these two scenarios at the SDM level to better understand the problem. Figure 5.3 illustrates the *state of the world* for Scenario 1 via a two-track approach; the bold arched arrows represent the understanding of the problem as it materialises onto the domain and application development tracks and the dashed line represents the *synchronisation points* between the two tracks. Though not obvious, the most noteworthy aspect of this diagram is the *natural alignment* between the two tracks and the problem domain.

Tight product interlocking

This property is made clearer by performing a similar exercise for Scenario 3, as does Figure 5.4. Even without a detailed analysis, it should be noticeable that the picture becomes *considerably more complex*; the previous natural alignment now gives way to a far more intricate set of relationships — many of which bidirectional. The figure depicts the separation between the infrastructure problem domain and the *core* problem domain, and the effect each of these have on each other as the exploration of both domains takes

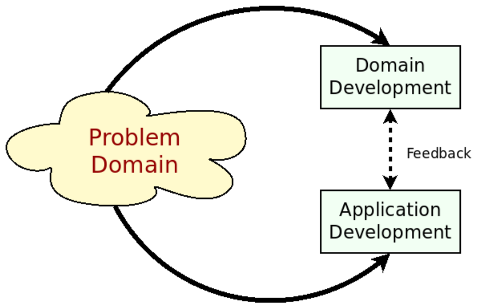


Figure 5.3: Typical MDE application.

place. In hindsight, the increase in complexity should not be surprising because, as the scope of the infrastructural domain grows, it becomes a software product in its own right. Thus, there is an attempt to simultaneously engineer *two tightly interlocked* software products, each already a non-trivial entity to start off with.

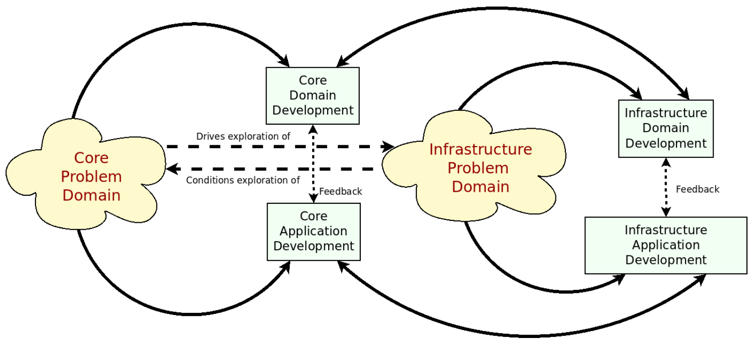

Core Problem Domain
Core Domain Development
Drives exploration of
Conditions exploration of
Feedback
Core Application Development
Infrastructure Problem Domain
Infrastructure Domain Development
Feedback
Infrastructure Application Development

Figure 5.4: MDE application with AC-MDSD.

At this juncture one may consider the ideal solution to be the use of vendor products as a way to insulate the problem domains. Unfortunately, experimental evidence emphatically says otherwise, revealing that isolation may be necessary *but only up to a point*, beyond which it starts to become detrimental. We name this problem *over-generalisation*. In our opinion, what is lacking is the *deep synchronisation* needed between the two development tracks — an ingredient that Stahl *et al.* had already identified as being crucial to the success of the entire endeavour. On the other hand, adoption literature shows that this close collaboration can be found in abundance within in-house development, but there we suffer from the opposite problem of *under-generalisation*. That is, solutions are *too specific*, catering only for internal use cases.

Over-generalisation

Under-generalisation

What is called for is a highly cooperative relationship between infrastructure developers and end-users, in order to foster feature suitability — a relationship which is not directly aligned with traditional customer-supplier roles; but one which must also maintain a clear separation of roles and responsibilities — not the strong point of relationships between internal teams within a single organisation, striving towards a fixed goal. Any proposed approach must therefore aim to establish an *adequate* level of generalisation by mediating between these actors and their diverse and often conflicting agendas. We named this generalisation sweet-spot *barely general enough*, following on from

Barely general enough

Ambler's footsteps [Amb07][5], and created Figure 5.5 to place the dilemma in diagrammatic form.

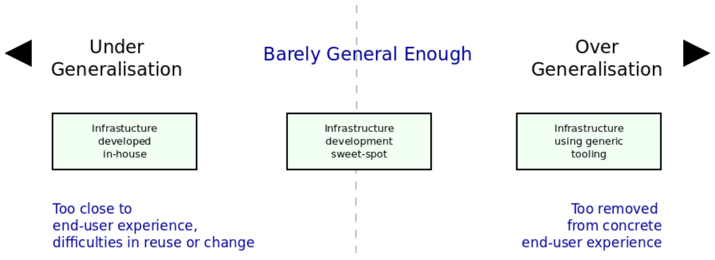

Under Generalisation
Barely General Enough
Over Generalisation
Infrastucture developed in-house
Infrastructure development sweet-spot
Infrastructure using generic tooling
Too close to end-user experience, difficulties in reuse or change
Too removed from concrete end-user experience

Figure 5.5: Different approaches to infrastructure development.

And now that the need for the generalisation has been made clear, we must change our focus towards the machinery needed to implement it. And, at this juncture, the *management of variability* takes centre stage.

5 Ambler states that (*emphasis ours*) "[. . .] if an artifact is just *barely good enough* then by definition it is at the most effective point that it could possibly be at." [Amb07]

6 MDE AND VARIABILITY MODELING

Despite their crucial importance, features are rarely modularized and there is only little support for incremental variation of feature functionality.

— Groher and Völter [GV07]

The employment of model-driven techniques for the engineering of software systems is often accompanied by a shift in focus from *individual* software products to *groups* of products with similar characteristics. This may happen tacitly — because the modeling process reveals these commonalities as abstractions (*cf.* Section 3.1) and good software engineering practices such as modularity and reuse create the conditions for their sharing across products — or by explicit design.

Motivation

Whichever its origins, this type of engineering presents challenges of a *different kind*, as Stahl *et al.* note:

> [...] MDSD often takes place not only as part of developing an entire application, but in the context of creating entire product lines and software system families. These possess very specific architectural requirements of their own that the architects must address. [Völ+13] (p. 5)

Thus, much stands to be gained by making the approach systematic, an aim often achieved in the literature by recourse to software diversity techniques such as variability modeling.[1]

The chapter is organised as follows. Section 6.1 introduces the notion of *product lines* and connects it to domain engineering. Variability proper is then tackled (Section 6.2), followed by a brief introduction on feature modeling (Section 6.3). The chapter then concludes with Section 6.4, where feature modeling is then integrated with MDE.

Chapter overview

6.1 SOFTWARE PRODUCT LINE ENGINEERING

The idea of abstracting commonalities between sets of programs has had a long history within computer science, starting as early as the 1970s with

1 For a comprehensive analysis on the state of the art in software diversity, see Schaefer *et al.* [Sch+12]. There, they defined software diversity as follows: "In today's software systems, typically different system variants are developed simultaneously to address a wide range of application contexts or customer requirements. This variation is referred to as software diversity."

Dijkstra's notion of *program families*.[2] A modern and systematised embodiment of Dijkstra's original insights can be found in Software Product Line Engineering (SPLE) [PBDL05; CN02], which is of particular significance to the present work because its principles and techniques are often employed in an MDE context [GV07; RR15; GV09].

Historical context

Pohl *et al.* define SPLE as "a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization." [PBDL05] Roth and Rumpe add that "SPLE focuses on identifying commonalities and variability (the ability to change or customize a system in a predefined way) to create components that are used to develop software products for an area of application." [RR15]

Definition

The mechanics of SPLE bring to mind Stahl *et al.*'s dual track process (Section 5.3), in that *Domain Engineering* is used to create a set of *core assets* and *Application Engineering* is applied to those core assets in order to develop a family of related Domain Engineering, \ product line} products, called a *product line* or a system family.[3] In this light, Domain Engineering defines the variation space available to all products within the product line, opening up possibilities in terms of variability, whereas the role of Application Engineering is to create specific *configurations* or *variants* for each product, reducing or eliminating variability. Figure 6.1 illustrates this idea.

{

The importance of managing variability within SPLE cannot be overstated, nor can the challenges of its management, as Groher and Völter make clear (*emphasis ours*): "The effectiveness of a software product line approach directly depends on *how well feature variability within the portfolio is managed* from early analysis to implementation and through maintenance and evolution." [GV07] The central question is then how to integrate a model-driven approach with the management of variability.

6.2 VARIABILITY MANAGEMENT AND VARIABILITY MODELS

Like the *program families* described in the previous section, variation itself has long been a going concern in software development; but, traditionally, it has been handled by programmatic means via techniques such as configuration

2 In [Dij70], Dijkstra states: "If a program has to exist in two different versions, I would rather not regard (the text of) the one program as a modification of (the text of) the other. It would be much more attractive if the two different programs could, in some sense or another, be viewed as, say, different children from a common ancestor, where the ancestor represents a more or less abstract program, embodying what the two versions have in common."

3 In Czarnecki's words (*emphasis his*) [Cza02]:

> *Domain engineering* (DE) is the systematic process of collecting, organizing, and storing past experience in building systems in a particular domain. This experience is captured in the form of reusable assets (*i.e.*, reusable work products), such as documents, patterns, reusable models, components, generators, and domain-specific languages. An additional goal of DE is to provide an infrastructure for reusing these assets (*e.g.*, retrieval, qualification, dissemination, adaptation, and assembly) during application engineering, *i.e.*, the process of building new systems. [...] Similar to the traditional single-system software engineering, DE also encompasses the three main process components of analysis, design, and implementation. In this context, however, they are referred to as *domain analysis*, *domain design*, and *domain implementation*."

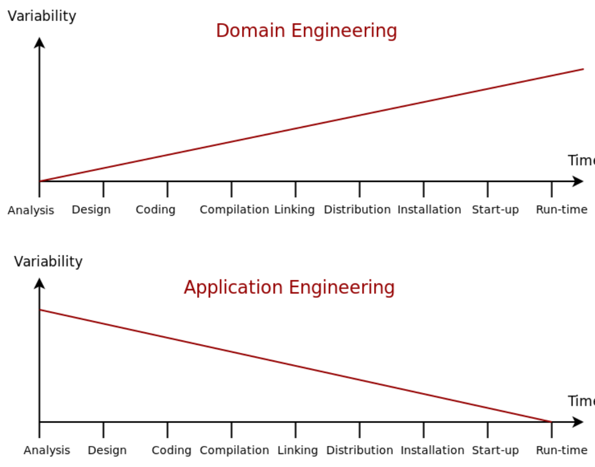


Figure 6.1: Variability management in time. *Source*: Author's drawing based on Bosch *et al.*'s image [Bos+01]

files, design patterns, frameworks and polymorphism — that is, at a low-level of abstraction — and scattered across engineering artefacts.[4] SPLE promotes instead *explicit variability management*, which Groher and Völter define as "the activity concerned with identifying, designing, implementing, and tracing flexibility in software product lines (SPLs)." [GV07] The idea is to promote variability to a first-class citizen within the engineering process.

Motivation

A variability management approach that resonates particularly with MDE is *variability modeling* — that is, the use of DSLs designed for expressing variability — for much the same reasons MDE promotes model use in general (*cf.* Section 3.1). Stoiber puts variability modeling in context:

> SPLE allows maximizing the reuse of commonality (*i.e.*, by developing all products on a common product platform) and of variability (*i.e.*, by a more modular development of variable functionality that can be added to or removed from the product more easily). This requires a variability model, though, to support an efficient specification and development of both the software product line as a whole and of individual application products. [Sto12]

Variability modeling

The benefits alluded to by Stoiber are more clearly identified by Czarnecki, who, in [Cza98] (p. 68), sees three main advantages resulting from the explicit modeling of variability:

- **Variability Abstraction**: By having a model of variability across the system — "a more abstract representation", in Czarnecki's words — it is now possible to reason about it independently of implementation mechanisms, which facilitates the work of Domain Engineering.

4 Czarnecki denounced this historically "inadequate modeling of variability", stating that "[the] only kind of variability modeled in current OOA/D is intra-application variability, *e.g.* variability of certain objects over time and the use of different variants of an object at different locations within an application."

Advantages of variability modeling

- **Variability Documentation**: From the perspective of Application Engineering, the variability space is made explicit and concise, therefore simplifying decisions around use and reuse.

- **Variability Traceability**: Engineers can also have a better understanding of the inclusion or exclusion of functionality because the variability model can answer those questions independently of the implementation.

Variability literature

These benefits help explain the abundance of variability modeling languages and notations in the literature, including AND/OR Tables [Mut+04], Decision Modeling [SRG11], Orthogonal Variability Modeling (OVM) [PBDL05], the Common Variability Language (CVL) [HWC13] and many others. A survey of all of these approaches lies beyond the scope of our work, given our needs — which only makes use of a set of high-level concepts in the variability domain — so we shall focus instead on only one approach: *feature modeling*.[5]

6.3 FEATURE MODELING

Feature modeling was originally introduced by Kang *et al.*'s work on Feature-Oriented Domain Analysis (FODA) [Kan+90] and subsequently extended by Czarnecki and Eisenecker [Cza+00], amongst many others.[6] As the name indicates, the concept central to their approach is the *feature*, which Groher and Völter define in the following manner: "[products] usually differ by *{* the set of features they include in order to fulfill *(sic.)* customer Features, relationship with product lines} requirements. A feature is defined as an increment in functionality provided by one or more members of a product line." [GV09] Features are thus are associated with product lines — each feature a cohesive unit of functionality with distinguishable characteristics relevant to a stakeholder[7] — and the interplay between features then becomes *itself* a major source of variability, as Groher and Völter go on to explain: "Variability of features often has widespread impact on multiple artifacts in multiple lifecycle stages, making it a pre-dominant (*sic.*) engineering challenge in software product line engineering."

Features and their relationships are captured by *feature diagrams* and *feature models*, as Czarnecki *et al.* tell us [CHE05a]: "A feature diagram is a tree of features with the root representing a concept (*e.g.*, a software system). Feature models are feature diagrams plus additional information such as feature

5 The interested reader is directed to Chen *et al.*'s [CABA09] systematic literature review of 34 approaches to variability management, which also provides a chronological background. In addition, Sinnema and Deelstra [SD07] authored a broad overview of the field, including surveys of DSLs and tooling, as well as performing a detailed analysis of six variability modeling approaches.

6 Feature orientation attracted interest even outside the traditional modeling community, giving rise to approaches such as Feature-Oriented Programming (FOP), which is "[...] the study of feature modularity and programming models that support feature modularity." [Bat03]

7 Note that we use the term *stakeholder* rather than customer or end user, taking the same view as Czarnecki *et al.* [CHE05a] (*emphasis ours*): "[...] we allow features with respect to *any stakeholder*, including customers, analysts, architects, developers, system administrators, etc. Consequently, a feature may denote *any* functional or non-functional characteristic at the requirements, architectural, component, platform, or any other level."

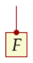
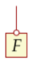
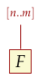
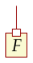
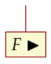
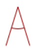
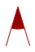
SYMBOL	EXPLANATION
	Solitary feature with cardinality $[1..1]$, *i.e.* mandatory feature
	Solitary feature with cardinality $[0..1]$, *i.e.* optional feature
	Solitary feature with cardinality $[n..m], n \geq 0 \wedge m \geq n \wedge m > 1$
	Grouped feature with cardinality $[0..1]$
	Feature model reference *F*
	Feature group with cardinality $\langle 1-1 \rangle$, *i.e. xor*-group
	Feature group with cardinality $\langle 1-k \rangle$, where *k* is the group size

Figure 6.2: Symbols used in cardinality-based feature modeling. *Source:* Author's drawing, based on Czarnecki and Helsen [CH06]

Feature diagrams, feature models

descriptions, binding times, priorities, stakeholders, *etc.*" Feature diagrams have found widespread use in the literature since their introduction, resulting on the emergence of several different extensions and variations.[8] For the purposes of the present chapter we shall make use of cardinality-based feature models, as described by Czarnecki *et al.* in [CHE05a] and whose notation Figure 6.2 summarises.

Example

The notation is perhaps made clearer by means of an example (Figure 6.3), which builds on from Figure 3.2 from Chapter 3. The top-most node of the feature diagram (*i.e.* `Car`) is called the *root feature*. Nodes `Body`, `Engine`, `Gear` and `Licence Plate` describe mandatory features whereas node `Keyless Entry` describes an optional feature. `Engine` contains a set of grouped features that are part of a *xor-group*, whereas `Gear` contains a set of features in a *or-group*. Or-groups differ from xor-groups in that they require that at least one feature from the group needs to be selected whereas xor-groups allow one and only one feature to be selected.

Solving

Feature diagrams have the significant property of being trivially convertible into Boolean logic formulas or to a Conjunctive Normal Form (CNF) representation, making them amenable to solving using established solvers such as Binary Decision Diagram (BDD) [CW07] and SAT [Bat05].

Importantly, feature modeling also has known shortcomings, and these were considered during our review of the literature. Most significant were those identified by Pohl *et al.* [PBDL05], namely that feature models mix the modeling of features with the modeling of variability and do not provide a way to segment features by intended destinatary — *i.e.* it is not possible to distinguish between features meant for internal purposes from those meant

8 An in-depth analysis of these variants would take too far afield with regards to the scope of the present work. The interested reader is directed to Czarnecki *et al.* [CHE05b], Section 2.2 (Summary of Existing Extensions), where a conceptual analysis of the main variants is provided.

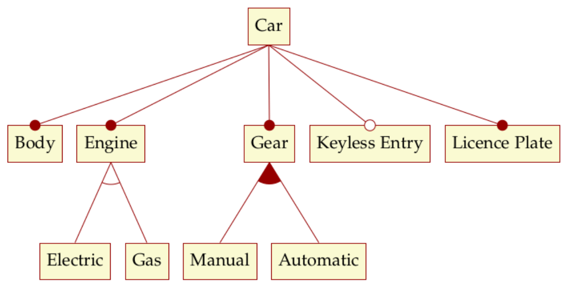


Figure 6.3: Sample feature model. *Source:* Author's drawing, modified from a Czarnecki and Wasowski diagram [CW07].

Limitations

for end-users. With OVM, Pohl *et al.* propose instead a decomposition of responsibilities. Clearly, there is validity to their concerns, as demonstrated by the fact that concepts that OVM brought into attention such as *variation points* — "delayed design decision[s]" [Bos+01] that "[. . .] allow us to provide alternative implementations of functional or non-functional features" as well as documentation — are now commonly used in the literature, even in the context of feature modeling. Nonetheless, since features provide an adequate level of granularity for our needs, we need not concern ourselves with Pohl *et al.*'s criticism. We do, however, require a clearer pictured of the relationship between feature models and the kinds of models that are typically found within MDE.

6.4 INTEGRATING FEATURE MODELING WITH MDE

Motivation

The crux of the problem is then on how to integrate MDES modeling techniques with variability management — or, more specifically for our purposes, with feature modeling. Clearly, having a feature model simply as a standalone artefact, entirely disconnected from the remaining engineering activities is just a form of MBE, as Czarnecki and Antkiewicz explain (*emphasis ours*): "Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, *features in a feature model are merely symbols*. Mapping features to other models, such as behavioral or data specifications, gives them semantics." [CA05]

Direct integration

Therefore, the availability of concise and interlinked representations of variability across models is a prerequisite to attain this semantically rich view of features. For their part, Groher and Völter argue that integrating variability directly within models has important advantages: "[. . .] due to the fact that models are more abstract and hence less detailed than code, variability on model level is inherently less scattered and therefore simpler to manage." [GV08] (*cf.* Figure 6.4).

Input variability

Whilst delving into the conceptual machinery of this integration, Groher and Völter [GV07; GV09] analysed the types of variability found in models and proposed dividing it into two kinds, *structural* and *non-structural*, defined as follows: "Structural variability is described using creative construction DSLs, whereas non-structural variability can be described using configuration languages." We name these two kinds *input variability* since they reflect

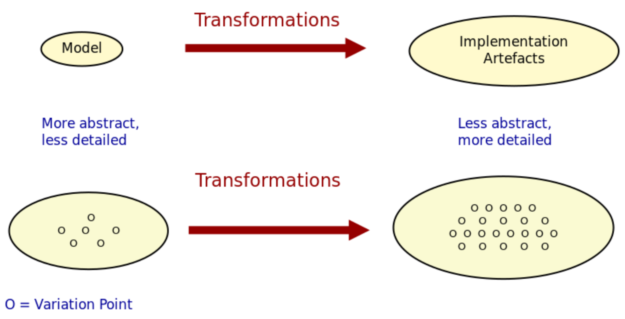


Figure 6.4: Mapping abstract models to detailed representations. *Source:* Author's drawing from Groher and Völter's image [GV08]

variation within the input models. In their view, the feature model becomes a metamodel for the product line[9], and their instances are the configuration models for products, with the final aim being to "[. . .] use a configuration model to define variants of a structural model." According to them, these variants can be generated in two ways:

- **Positive Variability**: The assembly of the variant starts with a small core, and additional parts are added depending on the presence or absence of features in the configuration model. The core contains parts of the model that are used by all products in the product line.

Generational variability

- **Negative Variability**: The assembly process starts by first manually building the "overall" model with all features selected. Features are then removed based on their absence from the configuration model.

Since these two types of variability are related to generation, we classify them as *generational variability*. Figure 6.5 illustrates these two techniques, applied to sample features A, B and C.

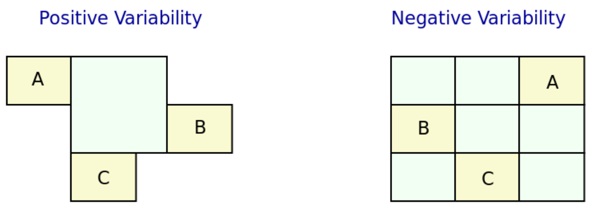


Figure 6.5: Positive and negative variability techniques. *Source:* Author's drawing based on images from Groher and Völter [GV09]

Given the cross-cutting nature of feature related concerns, Groher and Völter proposed using Aspect Oriented Programming (AOP) [Fil+04] techniques to implement positive and negative variability, to which they gave the perhaps overly-descriptive name of Aspect-Oriented Model Driven PLE (AO-MD-PLE). AO-MD-PLE has the advantage of considering all stages of software engineering, from problem space to solution space, including models, transformations (both M2M and M2T) and manually crafted code. In our opinion,

AO-MD-PLE

9 A view that aligns well with Czarnecki *et al.*'s idea of a feature model as the description of the set of all possible valid configurations within a system family [CHE05a].

its main downside is complexity, not only due to challenges inherent to AOP itself [CSS04; Ste06], but also because it uses several different tools to implement the described functionality and, understandably, requires changes at all levels of the stack.

UML Profiles

Undertakings of a less ambitious nature are also present in the literature. The simplest approach is arguably to integrate variability modeling directly with UML via a UML Profile, as suggested by Clauß's early work [Cla01], which focused on concepts such as variation points and variants. Ziadi *et al.* [ZHJ03] build on from this idea, expanding the focus to product line concepts. More recently, in [Pos+10; Pos+11], Thibaut *et al.* created a UML Profile for feature modeling concepts. Extending UML is advantageous due to its universal nature, but alas, it also inherits all of the challenges associated with the modeling suite.

Superimposed variants

Others have looked elsewhere. In [CA05], Czarnecki and Antkiewicz propose a template-based approach to map feature models to different kinds of models. There, they outline a technique of *superimposed variants*, in which a *model template* is associated with a feature model to form a *model family*. The model template is written in the DSL of the target model, and can be thought of as a superset of all possible models, containing model elements that are associated with features by means of *presence conditions*. Model templates can be instantiated given a feature configuration: "The instantiation process is a model-to-model transformation with both the input and output expressed in the target notation." The approach is reminiscent of Groher and Völter's positive variability, in that the template provides the overall model and MTs are then responsible for pruning unwanted model elements on the basis of the evaluation of presence conditions.

An interesting feature of superimposed variants are Implicit Presence Conditionss (IPCs):

IPCs

> When an element has not been explicitly assigned a PC by the user, an implicit PC (IPC) is assumed. In general, assuming a PC of true is a simple choice which is mostly adequate in practice; however, sometimes a more useful IPC for an element of a given type can be provided based on the presence conditions of other elements and the syntax and semantics of the target notation.

IPCs example, limitations

IPCs facilitate the job of the modeler because they infer relationships between features and model elements based on a deep understanding of the underlying modeling language. For example, if two UML model elements are linked by an association and each element has a presence condition, a possible IPC is to remove both modeling elements if either of their presence conditions evaluates to false. Overall, Czarnecki and Antkiewicz's approach is extremely promising, as demonstrated by their prototype implementation, but in our opinion it hinges largely on the availability of good tooling. Asking individual MDE practitioners to extend their tools to support superimposed variants is not feasible due to the engineering effort required.

As part of our review of the literature we also investigated the application of variability management techniques to code generators. In [RR15], Roth and Rumpe motivate the need for the application of product line engineering techniques to code generation. Their paper provides a set of conceptual mechanisms to facilitate the product-lining of code generators, and outlines a useful set of requirements: "The main requirements for a code generator product line infrastructure are support for incremental code generation, specification of code generator component interfaces, support for validation of generated code, and support for individual semantics of a composition operator."

Requirements for SPLE code generators

For their part, Greifenberg *et al.* [Gre+16] reflected on the role of code generators within SPLE — particularly those that are implemented as product lines *themselves*: "[. . .] a code generator product is a SPL on its own, since it generates a variety of software products that are similar, and thus shares generator components potentially in different variants". Their work also introduces the concept of *variability regions*:

Code generators as product lines

> Variability regions (VRs) provide a template language independent approach to apply concepts of FOP [Feature-Oriented Programming (FOP)] to code generators. A VR represents an explicitly designated region in an artifact that has to be uniquely addressable by an appropriate signature.

Variability regions

Variability regions are accompanied by two DSLs: Layer Definition Language (LDL) and Product Configuration Language (PCL). The LDL is used to define relationships between variability regions, whereas the PCL defines individual configurations to instantiate variants. Variability regions and their modeling is certainly an interesting idea, but it is somewhat unfortunate that Greifenberg *et al.* did not link them back to feature models or to higher-level modeling in general.

LDL, PCL

Finally, Jörges' [Jör13] take on code generation, modeling and product lines is arguably the most comprehensive of all those analysed, given he advocates the development of code generators that take into account variant management and product lines as one of its core requirements [Jör13] (p. 8). *Genesys*, the approach put forward by Jörges in his dissertation, hinges on a service-oriented approach to the construction and evolution of code generators, anchored on the basis of models: "Both models and services are reusable and thus form a growing repository for the fast creation and evolution of code generators."

Genesys

Unfortunately, there were several disadvantages with his approach with regards to own purposes; namely, the reliance on a graphical notation for the design of code generators and, more significantly, the tool-specific nature of Genesys which cannot be considered outside of jABC.[10] As we have seen, these are in direct conflict with our own views on fitting with existing developer workflows rather than imposing new ones . Nonetheless, Jörges' work was very influential to our own, and we've carried across several features of his argument such as a clear outline of a set of requirements in order to guide the model-driven solution.

Genesys limitations

10 As per Jörges' [Jör13] (p. 43): "jABC is a highly customizable Java-based framework that realizes the tenets of XMDD [Extreme Model-Driven Development] [. . .] jABC provides a tool that allows users to graphically develop systems in a behavior-oriented manner by means of models called Service Logic Graphs (SLGs)."

At this juncture, we have now introduced all of MDE's basic building blocks required for the present work. In the next chapter we will turn our attention to our personal experiences with MDE, prior to learning about the discipline.

Part III

OUTLOOK

7 CONCLUSION

These studies highlight that the fundamentals of modeling – how designers 'do' abstraction, how engineers reason about a system in abstract terms, how organizations work with abstract concepts – are not well reflected in current modeling approaches. Indeed, the vast majority of modeling approaches – both industrial and academic – are developed without an appreciation for how people and organizations work.

— Whittle *et al.* [WHR14]

THIS MANUSCRIPT PROVIDED a high-level overview of core topics related to MDE theory which are of particular significance to our doctoral dissertation. Even taking into account only the presented material, it should be patently clear that MDE's achievements are immense from a theoretical standpoint. A large body of knowledge has been built and a great deal of light has been shed onto the *fundamental nature of modeling* in software systems, aptly summarised by Mussbacher *et al.* [Mus+14] (Section 2.1, "Major Areas of Advancement").

MDE's achievements

Nevertheless, the examination also highlighted a number of difficulties faced by the discipline. In our opinion, these can be summarised into two classes of contrasting but significant challenges. On one hand, too much of a reliance on empiricism and pragmatism puts the theoretical foundations into question. This was perhaps more of a danger in the early days of MDE, but it is one we should keep in mind. A second type challenge comes from the opposite direction: there is a danger that, over time, MDE's theory is becoming unmoored from its empirical and engineering roots and diffused across a vast body of knowledge. This is a problem faced by more mature disciplines, so it is perhaps instructive to listen to Von Neumann's words of warning (*emphasis his*):[1]

MDE's challenges

> As a mathematical discipline travels far from its empirical source, or still more, if it is a second and third generation only indirectly inspired by ideas coming from "reality" it is beset with very grave dangers. It becomes more and more purely aestheticizing, more and more purely *I'art pour I'art*. This need not be bad, if the field is surrounded by correlated subjects, which still have closer empirical connections, or if the discipline is under the influence of men (*sic.*) with an exceptionally well-developed taste. But there is a grave danger that the subject will develop along the line of least resistance, that the stream, so far from its source, will separate into a multitude of insignificant branches, and that the discipline

Von Neumann's Warning

1 Though he was talking specifically about mathematics, we believe his words also carry important lessons for fields such as MDE.

> will become a disorganized mass of details and complexities. In other words, at a great distance from its empirical source, or after much "abstract" inbreeding, a mathematical subject is in danger of degeneration. At the inception the style is usually classical; when it shows signs of becoming baroque, then the danger signal is up. [. . .]
>
> In any event, whenever this stage is reached, the only remedy seems to me to be the rejuvenating return to the source: the re-injection of more or less directly empirical ideas. I am convinced that this was a necessary condition to conserve the freshness and the vitality of the subject and that this will remain equally true in the future. [VN04]

Clearly, this very fine balancing act can only be attained via a careful marrying of theory with application. In this spirit, we are led to conclude that any application of MDE must ensure it takes into account practical experience on the field, rather than just theoretical analysis.

Part IV

ANNEX

A
CONCEPT MAP

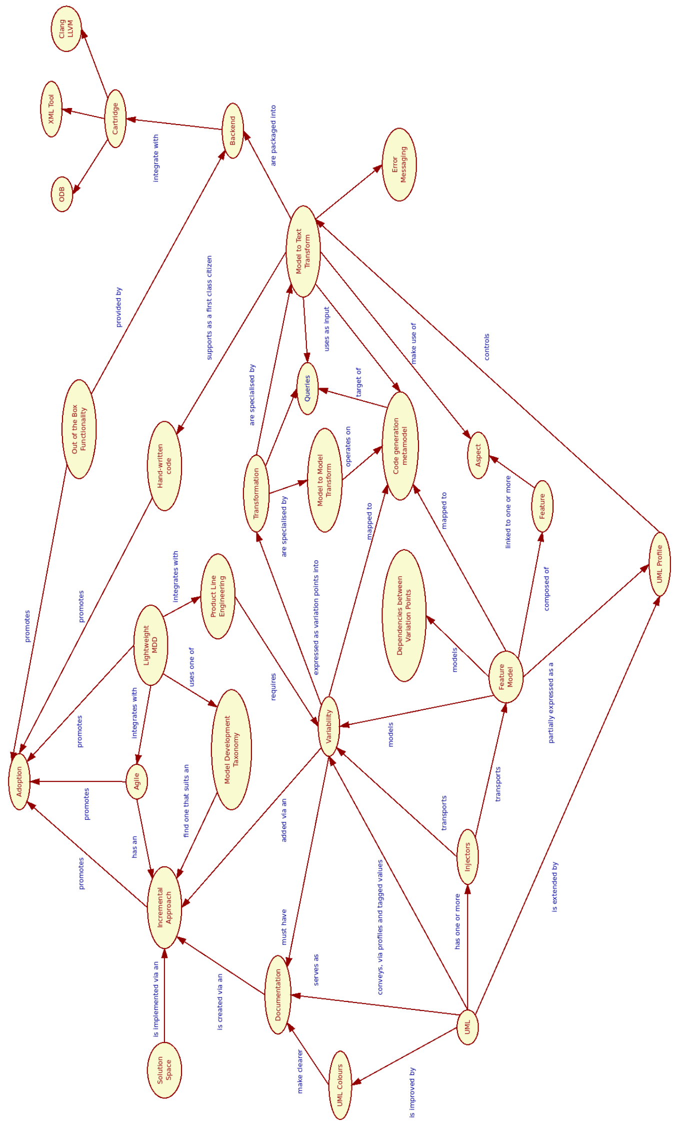


Figure A.1: Concept map for the Dogen domain

BIBLIOGRAPHY

[ÁES01] José M Álvarez, Andy Evans, and Paul Sammut. "Mapping between levels in the metamodel architecture". In: *International Conference on the Unified Modeling Language*. Springer. 2001, pp. 34–46 (cit. on p. 19).

[AF03] David E Avison and Guy Fitzgerald. "Where now for development methodologies?" In: *Communications of the ACM* 46.1 (2003), pp. 78–82 (cit. on p. 42).

[AF16] Sabah Al-Fedaghi. "Function-behavior-structure model of design: an alternative approach". In: *Int. J. Adv. Comput. Sci. Appl* 7.7 (2016), pp. 133–139 (cit. on p. 17).

[AK02] Colin Atkinson and Thomas Kühne. "Profiles in a strict metamodeling framework". In: *Science of Computer Programming* 44.1 (2002), pp. 5–22 (cit. on p. 20).

[AR08] Mohsen Asadi and Raman Ramsin. "MDA-based methodologies: an analytical survey". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2008, pp. 419–431 (cit. on pp. 8, 43).

[Abr+04] Alain Abran et al. "Software engineering body of knowledge". In: *IEEE Computer Society, Angela Burgess* (2004) (cit. on p. 9).

[Amb07] Scott W Ambler. "Agile Model driven development (AMDD)". In: *XOOTIC MAGAZINE, February* (2007) (cit. on pp. 41, 43, 44, 48).

[Amb08] Scott W Ambler. "Agile software development at scale". In: *Balancing agility and formalism in software engineering*. Springer, 2008, pp. 1–12 (cit. on pp. 43, 44).

[Ame] David Ameller. "SAD: Systematic Architecture Design". PhD thesis. Universitat Politècnica de Catalunya (cit. on p. 9).

[Anj+12] Anthony Anjorin et al. "A framework for bidirectional model-to-platform transformations". In: *International Conference on Software Language Engineering*. Springer. 2012, pp. 124–143 (cit. on p. 35).

[Arc+11] Paolo Arcaini et al. "A model-driven process for engineering a toolset for a formal method". In: *Software: Practice and Experience* 41.2 (2011), pp. 155–166 (cit. on p. 25).

[Awa05] MA Awad. "A comparison between agile and traditional software development methodologies". In: *University of Western Australia* (2005) (cit. on p. 41).

[BBM05] Kathrin Berg, Judith Bishop, and Dirk Muthig. "Tracing software product line variability: from problem to solution space". In: *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. South African Institute for Computer Scientists and Information Technologists. 2005, pp. 182–191 (cit. on p. 30).

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Vol. 1. 1. Morgan & Claypool Publishers, 2012, pp. 1–182 (cit. on pp. 3, 8–11, 15, 19–21, 23, 32, 33, 43).

[BF+14] Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014 (cit. on p. 9).

[BL13] Omar Badreddin and Timothy C Lethbridge. "Model oriented programming: bridging the code-model divide". In: *Proceedings of the 5th International Workshop on Modeling in Software Engineering*. IEEE Press. 2013, pp. 69–75 (cit. on pp. 10, 26).

[Bat03] Don Batory. "A tutorial on feature oriented programming and product-lines". In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE. 2003, pp. 753–754 (cit. on p. 54).

[Bat05] Don Batory. "Feature models, grammars, and propositional formulas". In: *International Conference on Software Product Lines*. Springer. 2005, pp. 7–20 (cit. on p. 55).

[Béz+03] Jean Bézivin et al. "First experiments with the ATL model transformation language: Transforming XSLT into XQuery". In: *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. Vol. 37. 2003 (cit. on p. 32).

[Béz04] Jean Bézivin. "In search of a basic principle for model driven engineering". In: *Novatica Journal, Special Issue* 5.2 (2004), pp. 21–24 (cit. on p. 19).

[Béz05a] Jean Bézivin. "Model driven engineering: An emerging technical space". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2005, pp. 36–64 (cit. on pp. 8, 11, 17, 32).

[Béz05b] Jean Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188 (cit. on pp. 16, 19, 21, 26, 35, 36).

[Boo18] Boost. *Boost Spirit*. 2018. URL: https://www.boost.org/doc/libs/1_68_0/libs/spirit/doc/html/index.html (visited on 11/06/2018) (cit. on p. 25).

[Bos+01] Jan Bosch et al. "Variability issues in software product lines". In: *International Workshop on Software Product-Family Engineering*. Springer. 2001, pp. 13–21 (cit. on pp. 53, 56).

[Bri96] Sjaak Brinkkemper. "Method engineering: engineering of information systems development methods and tools". In: *Information and software technology* 38.4 (1996), pp. 275–280 (cit. on p. 42).

[Bru+18] Jean-Michel Bruel et al. "Model Transformation Reuse Across Metamodels". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2018, pp. 92–109 (cit. on p. 23).

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. "Mapping features to models: A template approach based on superimposed variants". In: *International conference on generative programming and component engineering*. Springer. 2005, pp. 422–437 (cit. on pp. 56, 58).

[CABA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. "Variability management in software product lines: a systematic review". In: *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University. 2009, pp. 81–90 (cit. on p. 54).

[CH06] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–645 (cit. on pp. 21, 22, 30, 55).

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Formalizing cardinality-based feature models and their specialization". In: *Software process: Improvement and practice* 10.1 (2005), pp. 7–29 (cit. on pp. 54, 55, 57).

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Staged configuration through specialization and multilevel configuration of feature models". In: *Software Process: Improvement and Practice* 10.2 (2005), pp. 143–169 (cit. on p. 55).

[CN02] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Vol. 3. Addison-Wesley Reading, 2002 (cit. on p. 52).

[CSS04] Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Stoerzer. "AOP considered harmful". In: *1st European Interactive Workshop on Aspect Systems (EIWAS)*. 2004 (cit. on p. 58).

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. "Feature diagrams and logics: There and back again". In: *Software Product Line Conference, 2007. SPLC 2007. 11th International*. IEEE. 2007, pp. 23–34 (cit. on pp. 55, 56).

[Cla01] Matthias Clauß. "Generic modeling using UML extensions for variability". In: *Workshop on Domain Specific Visual Languages at OOPSLA*. Vol. 2001. 2001 (cit. on p. 58).

[Cue16] César Cuevas Cuesta. "Metaherramientas MDE para el diseño de entornos de desarrollo de sistemas distribuidos de tiempo real". PhD thesis. Universidad de Cantabria, 2016 (cit. on p. 7).

[Cza+00] Krzysztof Czarnecki et al. "Generative programming and active libraries". In: *Generic Programming*. Springer, 2000, pp. 25–39 (cit. on pp. 16, 54).

[Cza02] Krzysztof Czarnecki. "Domain engineering". In: *Encyclopedia of Software Engineering* (2002) (cit. on p. 52).

[Cza98] Krzysztof Czarnecki. *Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models*. Computer Science Department, Technical University of Ilmenau, 1998 (cit. on pp. 11, 53).

[DBC88] Alan M. Davis, Edward H. Bersoff, and Edward R. Comer. "A strategy for comparing alternative software development life cycle models". In: *IEEE Transactions on software Engineering* 14.10 (1988), pp. 1453–1461 (cit. on p. 41).

[DS92] Charles Donnelly and Richard M Stallman. *Bison 1.20: the YACC-compatible parser generator*. Free Software Foundation, 1992 (cit. on p. 25).

[Des14] Mihai Liviu Despa. "Comparative study on software development methodologies". In: *Database Systems Journal* 5.3 (2014), pp. 37–56 (cit. on p. 41).

[Dij70] Edsger Wybe Dijkstra. *Notes on structured programming*. 1970 (cit. on p. 52).

[Dis+14] Zinovy Diskin et al. "Towards a rational taxonomy for increasingly symmetric model synchronization". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2014, pp. 57–73 (cit. on p. 30).

[EB10] Moritz Eysholdt and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2010, pp. 307–309 (cit. on p. 25).

[Ell04] Geoffrey Elliott. *Global business information technology: an integrated systems approach*. Pearson Education, 2004 (cit. on pp. 41, 42).

[Est+07] Jeff A Estefan et al. "Survey of model-based systems engineering (MBSE) methodologies". In: *Incose MBSE Focus Group* 25.8 (2007), pp. 1–12 (cit. on pp. 10, 41–43).

[Eva04] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley, 2004. ISBN: 0321125215 9780321125217 (cit. on p. 16).

[FR07] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap". In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, pp. 37–54 (cit. on pp. 12, 13, 26, 29, 30, 34).

[Fav04] Jean-Marie Favre. "Towards a basic theory to model model driven engineering". In: *3rd workshop in software model engineering, wisme*. Citeseer. 2004, pp. 262–271 (cit. on pp. 7, 8).

[Fil+04] Robert Filman et al. *Aspect-oriented software development*. Addison-Wesley Professional, 2004 (cit. on p. 57).

[Fow04] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004 (cit. on p. 11).

[GHN10] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. "Model synchronization at work: keeping SysML and AUTOSAR models consistent". In: *Graph transformations and model-driven engineering*. Springer, 2010, pp. 555–579 (cit. on pp. 8, 30).

[GSD09] Gábor Guta, Wolfgang Schreiner, and Dirk Draheim. "A lightweight mdsd process applied in small projects". In: *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*. IEEE. 2009, pp. 255–258 (cit. on pp. 43, 44).

[GV07] Iris Groher and Markus Voelter. "Expressing feature-based variability in structural models". In: *In Workshop on Managing Variability for Software Product Lines*. Citeseer. 2007 (cit. on pp. 51–53, 56).

[GV08] Iris Groher and Markus Voelter. "Using Aspects to Model Product Line Variability." In: *SPLC (2)*. 2008, pp. 89–95 (cit. on pp. 56, 57).

[GV09] Iris Groher and Markus Voelter. "Aspect-oriented model-driven software product line engineering". In: *Transactions on aspect-oriented software development VI*. Springer, 2009, pp. 111–152 (cit. on pp. 29, 52, 54, 56, 57).

[Gre+15a] Timo Greifenberg et al. "A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages". In: *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*. IEEE. 2015, pp. 74–85 (cit. on p. 31).

[Gre+15b] Timo Greifenberg et al. "Integration of handwritten and generated object-oriented code". In: *International Conference on Model-Driven Engineering and Software Development*. Springer. 2015, pp. 112–132 (cit. on p. 31).

[Gre+16] Timo Greifenberg et al. "Modeling variability in template-based code generators for product line engineering". In: *arXiv preprint arXiv:1606.02903* (2016) (cit. on p. 59).

[Gro14] Object Management Group. *MDA Guide Version 2.0*. 2014 (cit. on p. 33).

[HLR08] Thomas Hettel, Michael Lawley, and Kerry Raymond. "Model synchronisation: Definitions for round-trip engineering". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 31–45 (cit. on p. 30).

[HS+13] Brian Henderson-Sellers et al. "Ptolemaic Metamodelling?: The Need for a Paradigm Shift". In: *Progressions and Innovations in Model-Driven Software Engineering*. IGI Global, 2013, pp. 90–146 (cit. on pp. 18–20).

[HSB12] Brian Henderson-Sellers and Arjan Bulthuis. *Object-oriented metamethods*. Springer Science & Business Media, 2012 (cit. on pp. 18, 19).

[HWC13] Øystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. "CVL: common variability language." In: *SPLC*. 2013, p. 277 (cit. on p. 54).

[Har17] Robert Harper. *What, if anything, is a programming paradigm?* 2017 (cit. on p. 8).

[Hut+11] John Hutchinson et al. "Empirical assessment of MDE in industry". In: *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE. 2011, pp. 471–480 (cit. on p. 10).

[JBF09] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. "Model driven language engineering with kermeta". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2009, pp. 201–221 (cit. on p. 23).

[Joh+75] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ, 1975 (cit. on p. 25).

[Jör13] Sven Jörges. *Construction and evolution of code generators: A model-driven and service-oriented approach.* Vol. 7747. Springer, 2013 (cit. on pp. 11, 17, 19, 23, 24, 31, 59).

[Jou+08] Frédéric Jouault et al. "ATL: A model transformation tool". In: *Science of computer programming* 72.1-2 (2008), pp. 31–39 (cit. on p. 23).

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. "Technological spaces: An initial appraisal". In: *CoopIS, DOA* 2002 (2002) (cit. on p. 31).

[KK84] Jeffrey E Kottemann and Benn R Konsynski. "Dynamic Metasystems for Information Systems Development." In: *ICIS*. 1984, p. 14 (cit. on pp. 18, 19).

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. "Toward an engineering discipline for grammarware". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14.3 (2005), pp. 331–380 (cit. on p. 25).

[KPP08] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. "The epsilon transformation language". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 46–60 (cit. on p. 23).

[Kan+90] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990 (cit. on p. 54).

[Kur07] Ivan Kurtev. "State of the art of QVT: A model transformation language standard". In: *International Symposium on Applications of Graph Transformations with Industrial Relevance*. Springer. 2007, pp. 377–393 (cit. on p. 23).

[Lan+18] Kevin Lano et al. "Technical Debt in Model Transformation Specifications". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2018, pp. 127–141 (cit. on p. 23).

[MA07] Parastoo Mohagheghi and Jan Aagedal. "Evaluating quality in model-driven engineering". In: *Modeling in Software Engineering, 2007. MISE'07: ICSE Workshop 2007. International Workshop on*. IEEE. 2007, pp. 6–6 (cit. on p. 24).

[MBFBJ02] Stephen J Mellor, Marc Balcer, and Ivar Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on p. 30).

[MMP10] Ole Lehrmann Madsen and Birger Møller-Pedersen. "A unified approach to modeling and programming". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2010, pp. 1–15 (cit. on pp. 26, 44).

[MVG06] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142 (cit. on pp. 21, 31).

[Mat11] Reza Matinnejad. "Agile model driven development: An intelligent compromise". In: *Software Engineering Research, Management and Applications (SERA), 2011 9th International Conference on*. IEEE. 2011, pp. 197–202 (cit. on pp. 8, 11, 43, 44).

[Mey88] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall New York, 1988 (cit. on p. 10).

[Mus+14] Gunter Mussbacher et al. "The relevance of model-driven engineering thirty years from now". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2014, pp. 183–200 (cit. on pp. 8, 12, 65).

[Mut+04] Dirk Muthig et al. "GoPhone-a software product line in the mobile phone domain". In: *IESE-Report No* 25 (2004), pp. 1–104 (cit. on p. 54).

[Neu16] Patrick Neubauer. "Towards Model-Driven Software Language Modernization." In: *STAF Doctoral Symposium/Showcase*. 2016, pp. 11–20 (cit. on pp. 25, 35).

[OMG17] Object Management Group OMG. "Unified Modeling Language (UML) 2.5.1 Specification". In: *Final Adopted Specification (December 2017)* (2017) (cit. on pp. 23, 42).

[PBDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005 (cit. on pp. 11, 52, 54, 55).

[PKP12] Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. "Metamodelling for grammarware researchers". In: *International Conference on Software Language Engineering*. Springer. 2012, pp. 64–82 (cit. on p. 25).

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013 (cit. on p. 25).

[Pod17] Karlis Podnieks. "Philosophy of Modeling: Some Neglected Pages of History". In: (2017) (cit. on pp. 15, 16).

[Poo01] John D Poole. "Model-driven architecture: Vision, standards and emerging technologies". In: *Workshop on Metamodeling and Adaptive Object Models, ECOOP*. Vol. 50. Citeseer. 2001 (cit. on p. 10).

[Pos+10] Thibaut Possompès et al. "A UML Profile for Feature Diagrams: Initiating a Model Driven Engineering Approach for Software Product Lines". In: *Journée Lignes de Produits*. 2010, pp. 59–70 (cit. on p. 58).

[Pos+11] Thibaut Possompès et al. "Design of a UML profile for feature diagrams and its tooling implementation". In: *Software Engineering & Knowledge Engineering*. 2011, pp. 693–698 (cit. on p. 58).

[RP08] Raman Ramsin and Richard F Paige. "Process-centered review of object oriented software development methodologies". In: *ACM Computing Surveys (CSUR)* 40.1 (2008), p. 3 (cit. on p. 42).

[RR15] Alexander Roth and Bernhard Rumpe. "Towards product lining model-driven development code generators". In: *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*. IEEE. 2015, pp. 539–545 (cit. on pp. 52, 58).

[Rot+89] Jeff Rothenberg et al. "The nature of modeling". In: *in Artificial Intelligence, Simulation and Modeling* (1989) (cit. on p. 16).

[SD07] Marco Sinnema and Sybren Deelstra. "Classifying variability modeling techniques". In: *Information and Software Technology* 49.7 (2007), pp. 717–739 (cit. on p. 54).

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. "A comparison of decision modeling approaches in product lines". In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. ACM. 2011, pp. 119–126 (cit. on p. 54).

[SS11] Fredrik Seehusen and Ketil Stølen. "An evaluation of the graphical modeling framework (gmf) based on the development of the coras tool". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2011, pp. 152–166 (cit. on p. 25).

[SVC06] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN: 0470025700 (cit. on pp. 10, 17).

[SVI12] IM SVITS. "A Comparative Analysis of Different types of Models in Software Development Life Cycle". In: *International Journal* 2.5 (2012) (cit. on p. 41).

[Sch+12] Ina Schaefer et al. *Software diversity: state of the art and perspectives*. 2012 (cit. on p. 51).

[Sei03] Edwin Seidewitz. "What models mean". In: *IEEE software* 20.5 (2003), pp. 26–32 (cit. on p. 19).

[Spr04] Jonathan Sprinkle. "Model-integrated computing". In: *IEEE potentials* 23.1 (2004), pp. 28–30 (cit. on p. 10).

[Sta73] Herbert Stachowiak. "General model theory". In: *Springer* (1973) (cit. on pp. 15, 16, 31).

[Ste06] Friedrich Steimann. "The paradoxical success of aspect-oriented programming". In: *ACM Sigplan Notices*. Vol. 41. 10. ACM. 2006, pp. 481–497 (cit. on p. 58).

[Ste+09] David Steinberg et al. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009 (cit. on p. 25).

[Sto12] Reinhard Stoiber. "A new approach to product line engineering in model-based requirements engineering". PhD thesis. Ph. D. thesis, University of Zurich, 2012 (cit. on p. 53).

[VN04] John Von Neumann. "The mathematician". In: *Musings of the Masters: An Anthology of Mathematical Reflections* (2004), p. 169 (cit. on p. 66).

[Völ09] Markus Völter. "MD* Best Practices". In: *Journal of Object Technology* 8 (2009), pp. 79–102 (cit. on p. 10).

[Völ+13] Markus Völter et al. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013 (cit. on pp. 3, 8, 10, 22, 24–27, 33–35, 43–45, 51).

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. "The state of practice in model-driven engineering". In: *IEEE software* 31.3 (2014), pp. 79–85 (cit. on pp. 9, 65).

[WJ05] Dennis Wagelaar and Viviane Jonckers. "Explicit platform models for MDA". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2005, pp. 367–381 (cit. on p. 35).

[Wag08] Dennis Wagelaar. “Composition techniques for rule-based model transformation languages”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 152–167 (cit. on p. 22).

[Was15] Charles S Wasson. *System engineering analysis, design, and development: Concepts, principles, and practices*. John Wiley & Sons, 2015 (cit. on p. 29).

[Whi+17] Jon Whittle et al. “A taxonomy of tool-related issues affecting the adoption of model-driven engineering”. In: *Software & Systems Modeling* 16.2 (2017), pp. 313–331 (cit. on p. 30).

[ZHJ03] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. “Towards a UML profile for software product lines”. In: *International Workshop on Software Product-Family Engineering*. Springer. 2003, pp. 129–139 (cit. on p. 58).

MASD

Model Assisted Software Development