Experience Report of Industrial Adoption of Model Driven Development in the Financial Sector

Marco Craveiro <marco.craveiro@gmail.com>

Abstract

This paper is an experience report that analyses the use of Model Driven Development (MDD) techniques to create and maintain a new software system for a large financial company. At the time of writing, the system has been in production for over nine years, providing a platform to investigate the long term effects of the approach. Software development was performed by a team of engineers lacking a theoretical background on MDD, which we termed *unorthodox practitioners*. Using a mix of qualitative methods, an exploration is performed on their experiences in developing production software with MDD via practical experimentation.

The paper has two main contributions. First, it provides more data on how MDD is used in industry, in particular by non-experts. Second, it employs prior analytical frameworks to analyse gathered data and extract insights on MDD adoption.

1 Introduction

MDD is an approach to software development that places the emphasis on *modeling* and *models*: it regards models as a primary artefact of the development process rather than an ancillary by-product. MDD enables the automatic generation of part or the entirety of a target system via the use of generators, which, through a process of refinement, "[..] systematically, and automatically transform abstract models to concrete code." [RR15].

Model Driven Engineering (MDE) is considered to be a superset of MDD since, in addition to taking on all of MDDs responsibilities, it also aims to tackle the broader scope of software engineering activities.[1]

MDE — and, by extension, MDD — is believed to have many benefits, including increased productivity and maintainability[2], and empirical studies have produced varying degrees of supporting evidence for these claims [Hut+11; And+14; SKSG07]. However, as Mohagheghi and Dehlen report [MD08], there is still no conclusive evidence that the promise of MDE holds for the general case. Furthermore, even though Whittle *et al.* demonstrated its widespread use across industry [WHR14], "MDE is arguably still a niche technology" [Mus+14] — particularly when compared to technologies of a similar lifespan such as Java, C# or, closer to home, the Unified Modeling Language (UML). Where it is practised, adherence to MDEs theoretical foundations varies widely, ranging from "orthodox" application to independent rediscovery.[3] The latter are of special interest to this paper, where they are termed *unorthodox practitioners*: these are developers working in the problem domain of MDE — modeling and transformations — but with little to no awareness of the existence of the discipline.

Clearly, it is important to continue to gather evidence of MDE's successes and failures, in academia as well as in industry, and to improve on prior experience in order to better understand what is hampering adoption. The objective of this paper is, then, threefold:

[1] Several authors accept this distinction between MDE and MDD, including Brambilla [BCW12] and Ameller [Ame] — subsequently re-enforced by Whittle *et al.* [WHR14]. Völter's groups them (and others) under the umbrella term *MD** [Völ09]. However, an authoritative taxonomy of modeling approaches, with rigorous definitions, could not be located. If indeed no such taxonomy exists, it would, perhaps, present a fruitful avenue for research.

[2] For one such claim, see Völter and Stahl in [Völ+13] — though, there, it is referred to as Model Driven Software Development (MDSD).

[3] Here, *orthodox application* is understood to mean MDE application according to the discipline's state of the art and best practices.

- **Supply More Evidence**: to accumulate additional empirical evidence of MDD adoption by presenting a description of an industrial product in the financial sector, used in production by a significant number of users.

- **Study Unorthodox Practitioners**: to provide details of MDD adoption by software developers that have not been exposed to MDD / MDE theory; to understand why no recourse was made to it; and to analyse what parts of the theory were rediscovered by the team.

- **Learn Lessons**: to deploy analytical frameworks defined in prior empirical studies in order to extract additional insights from gathered data; to identify a set of "lessons learned" and of *barriers to entry* obstructing MDE adoption.

The remainder of the paper is structured as follows. Section 2 presents a literature review of related empirical studies, highlighting where the present paper reuses or differs on the approach. Section 3 summarises the methods used to obtain and analyse data. Section 4 provides an overview of The Financial Company and of the problem space tackled by the new system, as well as a historical review of the project. Section 5 provides a mapping between internal concepts and MDD literature. Section 6 abridges the most salient points made by project personnel on the semi-structured interviews and is followed by Section 7, where an impact analysis is performed. Finally, Section 8 summarises the paper and proposes areas for future work.

2 Related Work

Mohagheghi and Dehlen provide a comprehensive meta-analysis of MDE adoption literature in [MD08]. Spanning 25 papers, their work sheds light on where and why MDE is applied, the maturity level of MDE — including the level of automation, software processes and tooling — and attempts to find evidence of the impact of MDE on productivity and software quality. Unfortunately, few of the underlying papers had clear impact evidence, leading them to state that "[f]uture work for evaluation of MDE should focus on performing more empirical studies, improving data collection and analyzing MDE practices so that success and failure factors and appropriate contexts for MDE can be better identified." The present paper was structured to address these concerns.

With [HRW11; Hut+11], Hutchinson *et al.* reportedly created the first systematic and multi-disciplinary empirical study on industrial practices and effectiveness of MDE, where they make use of a methodological mix that included questionnaire surveys, observational methods and semi-structured in-depth interviewing. Data obtained by semi-structured interviewing was employed to contextualise MDE adoption. Their study was instrumental in shaping the present work. Firstly, this paper makes use of a subset of their multi-disciplinary methods for data gathering, as well as reusing their approach of presenting both an organisational context and a personal context, as it paints a broader picture by taking into account non-technical issues. The organisational context is analysed using their organisational aspects framework. Secondly, this paper considers — and attempts to address — the key gaps they have identified, namely: "a lack of knowledge on how MDE is used in industry; a lack of understanding of how social factors affect MDE use; and a failure to assess aspects of MDE beyond UML, such as the benefits of code generation, domain-specific abstractions and model transformations."

Another very important factor in MDE adoption is tooling. In [Whi+13] — and, subsequently in [Whi+17] — Whittle *et al.* address this crucial issue. Their papers provide an insightful analysis of the state of tooling across a large sample population, but, more significantly, it defines a generalised taxonomy of tool-related considerations. Their taxonomy is applied to the MDD tooling of the project under study, forming the core of the impact analysis.

Additional papers were reviewed but lacked direct relevance to the present work. In [Mus+14], Mussbacher *et al.* point out a set of broad problems with MDE but then go on to tackle a much larger subject area, making their analysis less suitable for an empirical study such as the present. Andolfato *et al.* illustrate the application of MDE to Telescopes and Instrument Control Systems [And+14] but do not provide any analytical tooling we could

reuse. Similarly, Shirtz *et al.* produced a very detailed experience report on the well-structured and disciplined process of MDE adoption by a large financial organisation [SKSG07]. Unfortunately, the paper not was not relevant to our work given that we are specifically interested in studying the unstructured approaches of unorthodox practitioners. Both papers provide methodologies to measure improvement — an area of vital importance — but neither methodology was directly applicable to our case study.

In summary, the present paper is a direct response to findings on our literature review. On one hand, the review points out that there is a obvious need for additional studies that can be used to form a generalised argument about benefits and problems with MDE adoption, and for those studies to avoid previously identified deficiencies. On the other hand, a general pattern emerged from the literature: there is a focus on the usage of MDE in the context of its theoretical foundations.[4] Given the "niche" status of MDE in industry, it seems relevant to report on the experiences of unorthodox practitioners.

3 Study Method

This paper employs three different qualitative methods for data collection, described below.

Observational Methods The author was a member of the development team of the project under study, for three years, starting at its inception. Thus, the paper makes use of personal notes and recollections at this initial stage, via participatory observation. Moreover, the author was on site for another two years at the latter stages of the project, allowing a non-participatory observation of the evolution of personnel, processes and organisational structure.

To counter the limitations of these methods — such as personal bias, incorrect or missing recollections and so forth — developers present at the time were asked to review the paper.

Semi-Structured Interviewing A number of one-to-one interviews was carried out with relevant personnel. This approach was chosen for the reasons outlined by Hutchinson *et al.* in [Hut+11]: a) it encouraged interviewees to reflect on their expertise and experiences; b) it allowed exploring in greater detail aspects of interest regarding their MDD experiences; and c) it captured the practical realities of MDD deployment.

The interview data was sourced from recorded audio, subsequently transcribed, lasting between twenty to forty minutes. A total of three interviews was made. In order to benefit from a wider range of views, interviewees were picked from both the early and latter stages of the project, and spanned different levels of the organisational hierarchy. The questions followed a core structure, but interviewees were allowed to deviate according to their interests and recollections.

Internal Documentation The company under study allowed access to documents detailing some of the earlier design decisions, which provided useful insights to the motivations of the participants at the time the decisions were made.

4 The Financial Company

This section introduces the requirements of The Financial Company for a new software system, describes its architecture, explains how models and code generation were used and provides a brief historical overview of the system's development. Local terminology is employed throughout this section in the interest of accuracy, but Section 5 subsequently maps it to MDD.

4.1 Requirements

Around 2010 (year zero), The Financial Company made a business-driven decision to rewrite Old System, one of its real-time systems. Several strategic objectives were to be attained in New System; we shall focus only on those which are deemed to be connected with the adoption of MDD.

Architectural Cleanup Whilst Old System had a good separation between Client and Service tiers, its code base was monolithic and demanded a full redeploy with each release. In addition, services were stateful, communicated over a bespoke binary

[4] For instance, in [Hut+11], Hutchinson *et al.* specifically focus on sampling MDE practitioners for their study.

protocol and were tightly coupled to each other, as well as to the client. The client was very large — a *fat client* — and contained functionality which was now understood to belong to the service tier.

These architectural deficiencies were to be addressed by New System.

New Product Support Old System had served the company well in terms of the range of supported financial products, but it was now evident that augmenting it had become very difficult. Of course, such an undertaking has inherent complexity in any financial system, regardless of its design, as changes ripple across architectural layers — from UI to impacted services, to storage layer, to dependent internal and external systems. Nonetheless, due to its age and evolution, Old System imposed a great deal of accidental complexity[5], which made the process time consuming and demanding of both technical skill and experience with the code base. The manual coding required for type representation, serialisation, UI changes and the like was identified as a key bottleneck.

New System was to be designed with the explicit goal of allowing the business to add new products quickly, across all architectural layers.

Modern Technology Moving away from vendor legacy technology, which had reached its End-Of-Life, was a priority.

New System was to be designed with modern technology from the ground up.

Rapid and Iterative Delivery An Agile [Bec+01] based methodology was the chosen as the development process. The key objective was to enable a quick response to business requirements and to user feedback.

Where Old System required long release cycles, New System was to be released to production every two weeks, at the end of every Agile iteration.

Polyglot Programming New System was developed from the ground up but it did not exist in a vacuum; it inherited developers and technology from Old System, and was required to fit into the broader technological space of The Financial Company. As a result, some technology decisions were bounded by extraneous forces.

Significantly, New System had to support *polyglot programming*: "the activity of using several programming languages in a software system." [Fje08]. The external constraints that dictated this requirement can be summarised as follows. New System had to use an in-memory caching layer in Java as it was the prevalent caching technology within The Financial Company. In addition, given that the background of most available developers was in C#, the core part of the code base was to be written in C#. Finally, performance sensitive code from external teams had to be integrated with the system. Due to its nature, it was written in C++.

For these reasons, New System had to simultaneously support development in Java, C# and C++.

4.2 Architecture

The Architect anchored New System's architecture on two key pillars: *service-oriented*, as understood by the Service Oriented Architecture (SOA) principles [PL03]; and *document-centric*, following Representational State Transfer (REST) principles [FT00] (Chapter 5).

The choice of SOA was a direct response to the monolithic nature of Old System, and it was expected to help make the system easier to understand and maintain, and thus faster to release. REST was chosen because of its use of standard protocols — such as HTTP [Fie+99] — and open, standard document formats — such as XML and JSON — but also because it promoted the creation of stateless services, deemed to be easier to create and evolve, as well as more suitable for caching. All service interaction was to take place via documents, for requests as well as responses.

A similar approach was to be employed to make services self documentable: end users could target a well-defined Application Programming Interface (API) and metadata about the system would be made available via the very same open, standard document formats.

In New System, "everything was a document" — regardless of whether the content was data or metadata — and all interactions were to be request-response.

[5] Whittle *et al.* define *accidental complexity* as "[...] where the tools introduce complexity unnecessarily." [Whi+17]

4.3 The Model

After a period of prototyping, The Architect proposed placing a single document at the core of New System, called The Model, which would be responsible for describing all domain data types and their relationships. The Model only concerned itself with structural relationships, deliberately excluding any behavioural aspects.

Because The Model was itself a document, it too could be evolved and versioned, or even be supplied to end users. The Model would give transparency to developers, as they could consult it in order to understand the domain and its vocabulary — at least as far as data types were concerned — and they were to extend it as new use cases arrived. The Model was to be the *single source of truth* to describe the data types of the domain.

Additionally, The Model was also to be instrumental in addressing two core requirements: supporting polyglot programming and avoiding manual coding of trivial data types. This was to be achieved by using code generation to create programming language representations of The Model.

4.4 The Code Generator

The Code Generation Team was created and tasked with the implementing The Code Generator. The Code Generator's remit was confined to just The Model, with all other code in New System to be developed manually by the wider team.

The Code Generator had several important requirements. Firstly, it had to emit type representation, including arbitrary associations. Secondly, it had to support a well-defined set of serialisation formats, for all model types, across all supported programming languages — C#, C++ and Java. Thirdly, APIs were to be as symmetric as possible across languages, including for types such as collections and primitives as well as for reflection — needed, for example, to implement metadata APIs. Symmetry was believed to increase productivity when programming against The Model, as developers switched between programming languages. Lastly, The Code Generator's output was to be compiled into independent components that could be consumed by the rest of the system — *e.g.* an Assembly in C#, a JAR in Java and a DLL in C++. No further manual modification was allowed.

Like most New System code, The Code Generator was written in C# and used a template-based approach for text transformation. Microsoft's Text Template Transformation Toolkit (T4) was chosen as the templating language[6] due to its good integration with the existing tooling, active development by Microsoft and adequate feature set given the requirements.

The Model was represented by a XML document and described by a XML Schema Definition Language (XSD), and both were created and maintained manually, using Microsoft's Visual Studio Integrated Development Environment (IDE). The schema was code-generated into C#, using Microsoft's XSD Tool.[7] Code that was model independent was factored out into a manually developed helper library, which was referenced by the generated code. As a result of this approach, the metamodel was considered as a special kind of model and was not generated by the code generator itself — *i.e.* no *dog-fooding* was performed.

The Code Generation Team created a reference model to implement and document all of The Code Generator's features. The reference model was placed at the centre of a large test suite that verified the behaviour of the Code Generator, both in terms of the generated code as well as its performance and compatibility across supported languages.

Model versioning was handled by the team's Version Control System (VCS): The Model was checked in to version control, and all changes to The Model were performed as commits, subject to the same processes of code reviews, conflict handling and so forth as all other source code; generated code was checked in and used to validate changes to The Code Generator.

4.5 Project Evolution

Work on The Code Generator preceded New System development by around six months. During this period, The Architect and The Code Generation Team created and implemented a set of generic requirements that were deemed to be necessary in order to develop New System. These were devel-

[6] https://msdn.microsoft.com/en-us/library/bb126445.aspx

[7] https://docs.microsoft.com/en-us/dotnet/standard/serialization/xml-schema-definition-tool-xsd-exe

oped by making use of the reference model to test features.

The next eighteen months of development of New System were characterised by a rapid evolution of both The Code Generator and the surrounding system infrastructure. Given the choice of Agile as the development methodology, The Model, its XSD Schema and The Code Generator were all developed incrementally and symbiotically with the rest of New System: as developers requested features or reported bugs, The Code Generation Team acted accordingly by changing The Code Generator code, augmenting the reference model, updating The Model and/or extending its schema. Nevertheless, the fundamental architecture of The Code Generator — and of the generated code — remained that of the initial vision developed in isolation by The Code Generation Team and The Architect. As The Architect had envisioned, The Model and The Code Generator grew to become the centre of the New System, and found uses beyond core domain types, such as for configuration and messaging.

Around the two year mark, The Code Generator was deemed to be largely feature complete and The Code Generation Team was folded into New System's main development team. Evolution of The Code Generator largely ceased, other than trivial new features and bug-fixes. The business viewed New System as a very successful delivery, particularly with regards to the reliable two-week release cadence and the decrease on the cost of adding new products.

From year two to year eight (the present), the system has been under constant development and maintenance, albeit with a smaller development team. Deliveries tend to be largely business-focused, with a component of refactoring to tidy-up the system. Overall, New System has maintained its core architectural structure from its early phase, including having The Code Generator at its centre.

5 Technical Analysis

Whilst The Code Generation Team did not make use of standard MDE vocabulary, most of the concepts arrived at do have a straightforward correspondence to the canonical terms. Less obvious are the trade-offs made on key technical decisions, which perhaps require a broader understanding of the literature. We shall now provide a mapping back to core MDE concepts and highlight what we perceive as the most significant trade-offs.[8]

5.1 Metametamodel

In keeping with its bespoke nature, The Code Generator's metamodel was not described with existing metametamodel technology such as the Meta-Object Facility (MOF).[9] The metametamodel was discussed at length by The Architect and The Code Generation Team during the early development stages but, once the key ideas were understood and documented, it was left as an implicit construct and played no further role in development.

One of the most important consequences was that The Code Generation Team did not consider generating the metamodel using The Code Generator. Instead, the metamodel was treated as a special case and handled by external tooling. An opportunity was lost to dog-food The Code Generator but, as an advantage, the resulting code base was easier to understand by new developers.

5.2 Metamodel

As with the metametamodel, The Code Generator's metamodel was also developed internally, in an independent manner. Given that it factored out commonalities between the three target OO languages — *i.e.* Java, C++ and C# — its final shape resembled the structural aspects of Piefel's code-generation metamodel [Pie06]. Figure 1 contains a UML representation of part of the metamodel.

However, The Code Generation Team took the very significant decision of modeling only a well-defined subset of the structural aspects and only generating behaviours that are trivial functions of the structure such as serialisation. This simplified the scope of the metamodel and reduced the complexity of The Code Generator.

The run-time reflection layer also required access to metamodel concepts but it was implemented separately from The Code Generator's metamodel, resulting in very distinct APIs for each use case. This

[8] Note that the reader is expected to be familiar with the most common MDE terms. For an accessible introduction see [BCW12].

[9] http://www.omg.org/mof/

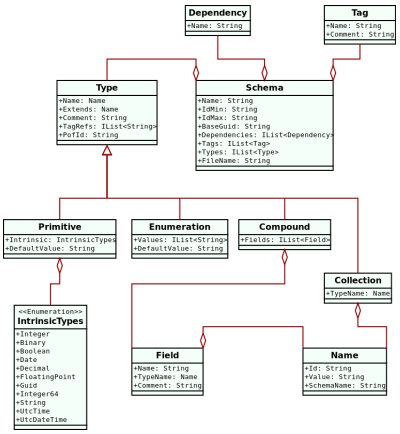

Figure 1: UML diagram of a metamodel fragment.

simplistic approach had the advantage of creating smaller and more focused APIs but the disadvantages were duplication of code and the need to learn two different APIs.

5.3 Model

Listing 33 contains a sample model `M1`, created for this paper, but which conforms to The Financial Company's XSD. The source was adapted to fit the space constraints.

```
<Schema Name="M1" IdMin="1" IdMax="10" BaseGuid="A...">
    <Dependencies>
        <Name Value="M2"/>
    </Dependencies>
    <Tags>
        <Name Value="T1"/>
        <Comment Text="Comment."/>
    </Tags>

    <Types xsi:type="Primitive">
        <Name Value="P1"/>
        <Id Value="1"/>
        <TagRefs Value="T1"/>
        <Intrinsic Value="Date"/>
        <Default Value="1970-01-01"/>
    </Types>

    <Types xsi:type="Compound">
        <Name Value="C1"/>
        <Id Value="2"/>
        <Extends Value="ModelValue"/>
        <TagRefs Value="T1"/>
        <Fields>
            <Name Value="F1"/>
            <TypeName Value="String" SchemaName="M2"/>
        </Fields>
        <Fields>
            <Name Value="F2"/>
            <TypeName Value="P1"/>
        </Fields>
    </Types>
</Schema>
```

Listing 1: Sample model conforming to the metamodel's XSD.

A common theme in MDE is to classify models according to the proximity to the implementation platform. Model Driven Architecture (MDA)[10] — Object Management Group's (OMG) particular vision of MDE — provides a useful distinction between Platform Independent Models (PIMs) and Platform Specific Models (PSMs), a classification that, according to Kurtev, "[...] is motivated by the constant change in implementation technologies and the recurring need to port software from one technology to another." [Kur07]

As the metamodel was designed to provide a thin abstraction above the implementation languages, technically its instance models are PIMs; these are then converted into PSMs during code generation. However, these concepts were left implicit in The Code Generator and encoded into the template instantiation process, which translates a Platform Independent Model (PIM) directly to source code. Whilst unsophisticated, the approach was sufficient to take advantage of the PIM / Platform Specific Model (PSM) separation, not only by having a single model for all three languages, but also by allowing upgrades to the Java and C# frameworks and to the C++ compiler — all performed via small changes to The Code Generator and with no impact to models. Nevertheless, given the limited scope for modeling in the project, the potential of the PIM / PSM separation was not exploited as fully as it could have otherwise been — for instance, by targeting architectural concepts such as SOA.

Similarly, the distinction between problem space and solution space was never made explicit, and thus many concepts in The Model either straddle both spaces or are purely implementation-level concepts. In part, this is a consequence of the design of the modeling language, closer to a General Purpose Modeling Language (GPML) rather than to a Domain Specific Language (DSL) purposely built for the problem space. Nevertheless, this pragmatic approach had advantages: developers understood intuitively how the modeling process worked as they saw it as a simple type translation between the description in The Model and the implementation in source code.

5.4 Variability

The Code Generator was designed explicitly with limited support for variability. It was understood

[10] http://www.omg.org/mda/

as a *special purpose* code generator, built exclusively for New System and with no application outside that remit. Therefore, The Code Generation Team took a strict approach, in keeping with the Agile methodology: in general, functionality was only introduced as needed and, if no longer required, it was to be removed from the system as quickly as possible.

As a result of this approach, the only significant variation point supported by The Code Generator was the ability to configure model element generation for each programming language. The main advantage of constraining variability so rigorously was the simplicity of the code base, but the disadvantages were not insignificant:

- as the code generator started before New System, a number of features were introduced speculatively; because there were no variation points, developers had to make use of these features even though they were not a good fit for the system;
- the lack of variability meant that The Code Generator was never considered for use on other systems, which perhaps could have benefited from this approach and could have shared the maintenance cost.

5.5 Transforms

Unsurprisingly, The Code Generator makes use of both Model-to-Model (M2M) and Model-to-Text (M2T) transforms, and there is a clear separation between these two major categories of transforms due to the use of a templating language for M2T.[11]

All M2M transformations were implemented by means of direct manipulation and written in imperative form in C#. As a result, the formalism around the metamodel and transformations were expressed in a limited manner, split between the facilities provided by XML tooling and manually written code in C#, and validated by an extensive test suite. This simplistic approach removed the need for scheduling as transforms were hard-coded to execute in a deterministic order and dependencies between transformations were implicitly understood by developers. In addition, as the metamodel was designed specifically for code generation, there was a limited need for M2M transforms other than model checking and validation.

M2T transforms were implemented as T4 templates. A minimalist Aspect Oriented (AO) framework was developed internally to allow for code reuse across templates. Given the limited variability needs, the role of aspects was correspondingly limited. As with transforms, aspects were manually added to templates by developers as required via C# code. This spirit of simplicity also guided decisions related to incrementality and directionality: there was no support for incrementality — models are processed from scratch on every execution of The Code Generator, and fully regenerated — and processing is performed in a strict one-way manner, from XML representation to source code. No manual changes are allowed to generated code.

Finally, tracing also followed a minimalist approach: an informal tracing strategy was adopted by adding comments to templates and aspects. Together with the knowledge of the code base, it was deemed sufficient given the requirements.

6 Personal Experiences

This section provides an insight into the individual experiences of software engineers, via extracts sourced from the semi-structured interviews. These have been edited for the sake of readability.

The section is divided into major themes, but there are limitations to this classification as they are closely interrelated — at times even overlapping — and do not have a natural order. In addition, whilst themes focus mostly on what could be construed as negative experiences, its important to note that all interviewees saw the role of the code generator as having both a positive and negative impact, but chose to focus more on negative aspects.

6.1 Ambitious Undertaking

Interviewees developed an appreciation for the difficulty of creating a general purpose code generator: "[...] It was a massively ambitious project, right? [To] [b]uild a general purpose code generator, is a very, very difficult thing."

[11] This section makes use of the feature model defined by Czarnecki and Helsen's in [CH06] for the analysis.

Once the magnitude of the task was understood, a natural process of de-scoping started to take place: "But when you say, 'I want to write a code generator that is going to work for everything', well then now you need to define what everything is. And how do you define *everything*? [...] You can't, so you say 'right I'm guessing I'm going to need lists, but I'm guessing I won't really need dictionaries, I'm guessing it will good enough just to have public setters but let's not worry about public/private, everything will be public and that's [...] something reasonable that I can write a code generator for, within a year and a half and [...] that'll have to do.' [...] [I]n reality, when you start using it, you find you need a whole load of other functionality because [...] your problem statement is so much bigger [...]."

In addition, resourcing was limited, which further curtailed ambitions: "[...] [The] C# programming language is a set of constraints, right? And you [...] do the best you can with them until they add some new language feature in. [...] Its just that [...] a code generator written in a year by three people is going to have many more constraints than a programming language written by hundreds of people across ten years."

The net result was a code generator that was feasible to implement, but which fell short of the original ambitions: "[W]e [...] wrote a general purpose code generator and tried to fit it around our requirements. But our general purpose code generator wasn't really rich enough to really do what we needed it to do, I don't think."

6.2 Evolving Purpose

Since the lowering of ambitions and the code generator's purpose are closely linked, its not surprising that interviewees revealed that their understanding of the code generator's purpose evolved over time. Originally, it was meant to produce a rich domain model, against which all of New System's code was to be written, as well as providing a serialisation mechanism between languages: "[W]e needed a way of sharing [...] information between these three tiers, [...] and have [...] the same model of what a *thing* is, in all three languages."

During this initial phase, many improvements were made: "[...] as we got experienced working with it, I think it was plain obvious that certain things could be improved relatively easily and those fed back into improvements [...] more into the code generated rather than the code generator itself actually."

However, over time, the conflicting nature of the requirements imposed on the code generator became apparent: "I think the biggest thing was more to do with... and its not, nothing inherent to code generation itself but I think it came from a steer that maybe the [The Code Generation Team] were given earlier on which was that it should be the same on all three languages. [...] And there are many ways of interpreting that requirement, I guess, and [...] the lead of the team seems to have taken a particularly [...] purist line on that, namely literally they were exactly the same, as close as you can get it, given the syntax of languages. And so you ended up with a model which was not particularly idiomatic in any of them."

Developers soon discovered that model types weren't suitable as rich domain types: "So what we ended up trying to do is, kinda leave the [...] going down into the model types to as late as possible in many cases. [...] So, effectively we ended up with a shadow model implemented — particularly in C#, less so in Java, because it doesn't do as much." This *shadow model* is a manually written, idiomatic version of the domain model, with associated helper code to translate from and to the code-generated model.

However, this approach was not used consistently. As a result, there is still a lack of clarity on the purpose of the code generator: "I don't know that that's the journey we had, though. I don't, I don't think that we've ever embraced that. It was [...] something we did in one place, to use it as a sort of a serialisation [...] helper [...] only. But I don't think necessarily that that is the right way [...] to use it anyway. Because [...] it almost defeats the purpose. If you have to, every time you create a [...] code generated type, you have to handcraft [...] an actual domain type, and then write an adaptor to the codegen'd type, then, well, why not just write a [...] serialiser for your domain type and leave out the middle man."

What is unmistakable is that there is a desire to minimise the scope and use of the code generator: "I think that, had it been used sparingly, and [...] in certain places where it was useful then [...] we might have got some benefits of it, like rapid devel-

opment without the [...] knock on problems that we eventually ended up with."

6.3 Speculative Features

Code generator development started prior to the development of the core of New System, meaning that The Code Generation Team and The Architect had to shape its initial direction in relative isolation. Trade-offs were made by taking this approach which may have not been obvious: "[...] [B]y the time we actually started using it in anger it was actually fairly mature. [...] In some ways that's a good thing, you want to start with something rather than nothing but [...] [i]t would have been possibly better to have had a less mature [...] product which we could, you know, look at and say 'no, this is not the right direction guys, lets change these principles.'"

As a result of this approach, a number of speculative features were added to The Code Generator. In many cases, these were found not to match the direction of New System: "[...] [T]here are a couple of design choices [...] there was an insistence on — this has nothing to do with code generation particularly, this is more to do with implementation — on them [the model elements] being [...] immutable. [...] Which is [...] useful in many [...] contexts but [...] we didn't have the problems that that sets out to solve, and it was imposed up front without actually finding out if we had any of these problems. [...] They were [...] kind of like a solution before we had a problem."

The issue was not restricted to immutability: "Early on someone decided we shouldn't have constructors, we should have everything with factory methods [...] I don't know where that idea comes from, it has been fashionable in Java for a while... I think sometimes C++ people choose it because of various quirks of that language, but it has never been a problem particularly in C#. Its certainly not idiomatic C#. [...] So, that's certainly quite hard to work with."

These and other speculative features were not optional, perhaps in order to restrict variability, so as a consequence software engineers started to make use of them best they could: "I think its a good point, you just learn to live with what you got, right? [...] These are the [...] constraints that we have, so we're going to have to [...] live with those constraints. [...] And you find a way, right?"

6.4 Unintended Consequences

As a result of all the factors discussed thus far, software engineers found creative workarounds for code generator deficiencies: "[...] [T]he problem was [...] [The Code Generator] only supported arrays [...]. So [...] if your code generated type is your main domain type, then your access patterns on it were problematic. We had lots of objects that were essentially key value stores, but it was a list of key values, and so we had to build stuff around that [...] if we wanted to get good access speeds, or we just accepted the fact that we were going to [...] search across an array every time we wanted to get a value for a key. [...] I think it was just a decision made earlier on, to only support a very limited number of collections, *i.e.* lists, and nothing else. [...] So [...] it was decided that [...] hash maps weren't, weren't necessary, dictionaries weren't necessary [...]. We worked around it in other places by treating [...] the code generated type as really just a serialisation type, so [...] [a] request would come in, we'd get deserialised into the codegen'd type and then, immediately after that, we'd get adapted into a [...] proper domain object."

When stacked together, all of the deficiencies of the code generator and the workarounds had consequences: "We lost any kind of [...] good domain modeling practices. There is no data encapsulation, data leaked all over the place, [...] our object model became a bit of a bag of stuff that got passed around everywhere and [...] it wasn't really encouraging good programming practices. [...] I think it [...] encouraged procedural programming, badly structured code and [...] no encapsulation."

However, a lot of code has already been written and it is difficult to address these problems: "In terms of [...] the future, [...] I think [...] its so entrenched in the system, I think its going to be really hard to get rid of it."

6.5 Business Alignment

A key point was the difficulty in justifying continued investment on a bespoke code generator from a business perspective: "Its quite product-y? So it almost feels like, you know, its something which [we] should be buying in or open source [...]. Not

what you want to be focusing your interest on, if you can [...] avoid it."

Over time, the budget of New System decreased as the system matured, and development became more focused on adding business functionality rather than on infrastructural changes. As a result, it became harder to justify further investment on the code generator: "It's kind of way down in the priority list."

Part of the difficulty is on how to show unambiguous benefits to the business: "Because unless you can see any real [...] [b]enefits [...] you're just going to work around [...] issues, rather than fixing them."

6.6 Rapid Development

Participants saw the code generator as an enabler of rapid development during New System's early days: "[...] [I]t did save a lot of time [...]. Because [...] we were adding huge numbers of types early on and [...] it didn't take that long to [...] write [...] some XML [...] describing [...] them [...] and have the types generated. And [...] you could generate thirty types in half an hour, which [...] had you hand crafted it and written all the serialisers that you needed, you would have been there [...] for days. So I think it did speed things up the early days [...]. And then probably slowed things down thereafter."

6.7 Interoperability Support

Code generation was also particularly successful at providing interoperability between languages: "[...] [B]eing able to create that once in the model and then have it generated for all of them [programming languages] and then knowing that it will compile, knowing that it will work, you can [...] then start sending these things around and then its essentially somebody else's problem, really, [...] that you send it in from the C# code and it pops up in the C++ code and [...] the C++ code will understand it, know what its got and more importantly it will have a strong type [...] its not [...] pulling things out of XML [...]. Now, at this point its now a rich object which they can now code against, safe in the knowledge that everything that should have been populated will have been populated. [...] And that, that was very useful [...]. And I must say, with this sort of three-technology [...] situation we had, it certainly made it an awful lot quicker [...]. The alternative would have been to, right we design something in C#, right, now you need to do the same thing in C++ and make sure that it all works, and you haven't forgotten something — dangling pointers and whatever else — and then make the same thing work again in Java, again make sure *that* works, make sure that they can all talk to each other and understand the same things in the same way [...] Codegen solved that problem for us. [...] So that became plumbing, and essentially we just say: 'this is what, you know, this class looks like' and everything will have the same understanding of what that class looks like."

7 Impact Analysis

In order to obtain a better picture of the role of MDD in the project under analysis, we shall use Whittle *et al.*'s "Taxonomy of Tool Related Considerations". [Whi+17] The taxonomy defines four different major groups of factors, each split into categories and subcategories. These are covered in the next four sections.

7.1 Technical Factors

We shall start by first considering the technical challenges and opportunities that were faced when applying MDD.

7.1.1 Tool Features

Developing an MDD tool internally is a key decision for any project, regardless of its size, and thus one which should not be taken lightly. Even though The Code Generation Team used off-the-shelf technologies for templating and XSD support, the development of The Code Generator was still a very large and complex undertaking — especially as they lacked field experience.

On one hand, the tool was designed specifically for the use cases requested by New System developers, which made it fit for purpose — although, as we saw, the understanding of its purpose has continually evolved. On the other hand, modeling was constrained by the tool's features and these

in turn were constrained by the modeling knowledge of the developers involved, both in The Code Generation Team and the wider New System team. Further: many crucial decisions were made — such as not modeling system behaviour, not supporting DSLs or not supporting the system's architecture — which are directly related to the perceived complexity in implementing these features. Thus the tool was simultaneously an enabler and a constrainer of modeling.

In addition, the development of features before their requirement was properly understood had a markedly negative impact on the usage of The Code Generator.

7.1.2 Practical Applicability

The Code Generator faced several scalability problems during its development, which were resolved by the team. For example, compilation times in C++ were deemed too high when building the entire model, as the code generator produced too many translation units. Changes were performed to address this issue, such as moving functionality to run-time via reflection, where performance was not significantly impacted. Serialisation was also aggressively optimised to meet New System's demands, both in terms of object sizes and total elapsed time — though there are still concerns about performance as the system load continually increases. In general, however, The Code Generator scaled well to a very large number of model elements (over one thousand).

Versioning was a challenge, both in terms of metamodel and model, as well as serialised objects from generated code. The solution was characteristically pragmatic: The Model's XML document was version-controlled in the same manner as source code — changes to the metamodel required manually updating models — and a serialisation format was designed to enable forward compatibility, and was used where required. Backwards compatibility was solved by not allowing the removal of fields, which can be marked as deprecated but cannot be deleted from The Model. Overall, the approach constrained model evolution as well as refactoring but was deemed sufficient for the needs.

With regards to the industrial quality of generated code and the flexibility and maturity of tooling, the approach taken was to evolve both generated code and tool to fit the requirements.

On one hand, there is a near unanimous agreement — as demonstrated in the Section 6 — that the quality of the generated code is inferior to manually crafted code and that it has had a negative impact on the overall quality of the system because it has encouraged a bad style of programming that eschews encapsulation. This is particularly problematic for new developers, who not only are more prone to making mistakes by following (and introducing) bad model-usage patterns but also need to learn The Code Generator's API; there is also a degree of tool subversion required in order to use the tool to the developer's benefit.

On the other hand, from a technical perspective the code has been used in a production system for several years, demonstrating the ability to withstand stresses and performing to high-availability standards, so one is forced to conclude that The Code Generator has industrial-grade quality as it was sufficient to enable the development of New System.

Finally, its important to mention that The Code Generator created a form of *lock-in*: New System is so dependent on it that it is difficult to make large changes to generated code without breaking large amounts of non-generated code. Thus, senior engineers tend to see it now as an *impediment* to progress, as opposed to an *enabler* of progress, as it was during the initial phase.

7.1.3 Complexity

It is clear that the tooling added a layer of complexity to New System, some of which was accidental in nature, but most of which can be attributed to the difficult domain of MDD. For example, it is difficult to find developers with enough technical knowledge to evolve the code generator.

In addition, and somewhat paradoxically, generated code was also responsible for adding complexity to New System. The style of programming was influenced by the generated code — large data types with little associated behaviour — resulting in a less encapsulated style of programming. This had very big implications for the overall complexity of the system, which evolved towards a procedural style instead of OO.

7.1.4 Human Factors

As with generated code and loss of abstraction, a case could also be made at the modeling level: developers started to think at the level of abstraction of The Code Generator. That is, the modeling process got skewed over time to "a way of doing things" that was supported by the code generator, which perhaps differs from how things would have otherwise be done. For example, the non-idiomatic properties of the generated code spread out to the handcrafted code which integrated with it.

Likewise, practices evolved to compensate for the usability constraints of the tool — such as a lack graphical representation, manual editing of XML and so forth — which may not be necessarily conducive to good modeling practices.

7.1.5 Theory

One of the biggest disadvantages of building your own tool by exploring the domain of modeling is how far you are from the state of the art in the literature; the weak theoretical foundations of the tool led to some of the problems described in this analysis. However, if the required amount of training was understood at inception, it is not clear that an MDD approach would have been undertaken at all. Thus there is a conflicting nature in the role of theory in tool design and application.

7.1.6 Impact on Development

MDD adoption had a very positive impact on development during the initial phase of problem space exploration because it was an enabler of rapid application development: it allowed for very quick prototyping, implementation and deployment of ideas across the stack, and to quickly refactor those ideas in response to user and stakeholder feedback.

Once the system matured, the impact is not seen as quite as positive. The unencapsulated style of development encouraged by the generated code resulted in a system that is harder to change because logic is now scattered across the code base, which acquired a procedural-like nature; in traditional OO development, objects would own their behaviour and mediate access to state, avoiding this scattering of responsibilities.

In addition, by enforcing a common API across programming languages, The Code Generator created a non-idiomatic style of programming that is unfamiliar to developers of all languages, and does not make efficient use of the native facilities available — for example, built-in reflection in C# and Java. As a result, new developers require a longer period of training, which includes both The Code Generator and the model's APIs.

7.2 Internal Organisational Factors

The alignment of The Financial Company's technology and business management structure was instrumental to the success of New System: they provided their full support and engaged actively in the Agile process. As discussed previously, the success of The Code Generator was closely connected to the success of New System because, through Agile, it permitted a fast exploration of a large problem domain, which would not have been economically feasible by manual means.

The organisational approach can be summarised as a set of key factors:[12]

- **Wholesale**: MDD processes were tightly coupled with Agile adoption from New System's early days and their removal was not considered. Nevertheless, there was also a very early demarcation of their role, thus a narrowing of scope for the impact of MDD. This mitigated most of the dangers associated by Hutchinson *et al.* with a wholesale approach [HRW11].
- **Iterative**: Whilst there was never a question of whether to use MDD or not, its application was largely an iterative process, evolved as the interactions between the wider team and The Code Generation Team became better defined.
- **Committed**: Management and developers were committed on making MDD work. This commitment was tested often during early development, as it was not understood if all necessary features could be implemented; management did not waver even under the difficult circumstances of these early years.
- **Business Led**: From the start, MDD adoption was justified as necessary to achieve business outcomes. The business was aware of all the work related to MDD and was supportive.

[12] These are as identified by Hutchinson *et al.* in [HRW11].

There are however, some factors pertaining to the internal organisation of The Financial Company which are not captured by this analysis.

First, the role of stakeholders impacted the system both positively and negatively. During the initial phase, stakeholders were supportive of the development of The Code Generator because its benefits were at its most obvious, given the rapid development of the system and the small size of the code base. However, during New System's mature phase, stakeholders became more reluctant to fund direct investment in The Code Generator because the short-term benefits are not clear from a business perspective and the work required is large due to the much larger size of the code base and the complexity of the system.

Secondly, and closely related, is the organisational structure chosen. It can be argued that management should not have merged The Code Generation Team with the main New System development team, but focused instead on improving The Code Generator. However, it was not clear how most of the required improvements would be addressed — particularly after the difficult experience of developing The Code Generator in the first place. Investing on a non-business product with an outcome that is difficult to predict is very hard to justify commercially.

A third criticism to the organisational approach is that there was an excessive focus on the low hanging fruit of structural modeling, which — when coupled with the strict containment of the use of code generation — curtailed New System's ability to fully benefit from MDD. However, as stated before, it is very hard to justify making a commitment to more advanced uses of MDD or MDE unless there is a obvious demonstration of its benefits; by the same token, it is very difficult to demonstrate its benefits until one starts to use it in a production system as was the case with The Financial Company.

Finally, it is important to address the issue of training. The Financial Company did not engage on any training aimed directly at MDD, even though it was conducted for a number of new technologies used in New System and deemed important by management. In our opinion, this is one of the biggest disadvantages of the unorthodox practitioner: without the required technical knowledge of the field, they do not see themselves as MDD practitioners — where there is an awareness of the discipline at all. This lack of visibility meant that training was not even considered as an option.

7.3 External Organisational Factors

The biggest external factor appears to be the niche status of MDD, and the lack of popularity of its tooling. In a team of over twenty software engineers, there was only passing knowledge of one tool: Eclipse's Eclipse Modeling Framework (EMF).[13] There is clear dearth of tooling, or at least of marketing of existing tools.

In addition, EMF was deemed too Java-specific and too complex to fit the requirements. Even though the tooling under the EMF umbrella has had great strides in usability, the barrier to entry is still too high for a casual developer that has not mastered MDD theory; there is a very large upfront investment required in mastering theory and tooling.

7.4 Social Factors

Social factors played a significant role in the adoption of MDD, both positively and negatively. In particular, the personalities involved promoted management structures which shaped the outcome: the separation between The Code Generation Team and the wider New System team helped create The Code Generator quickly but also reinforced separation between teams, where perhaps it would have been more beneficial to have integration.

In addition, developers ended up subverting the tool more than working with it, in order to compensate for its deficiencies, and whilst there was a high-level of trust in The Code Generator as far as serialisation was concerned, over time developers became wary of using it for domain modeling. At present there is an emphasis on using code generation "sparingly".

8 Conclusions

This paper produced a detailed analysis of the adoption of MDD by a financial company, where the development team did not have experience with the methodology, but instead rediscovered parts

[13] https://www.eclipse.org/modeling/emf

of it independently. These we termed *unorthodox practitioners*.

As with prior work, the present study has great difficulty in classifying the impact of MDD adoption as a uniquely positive or negative event. Hutchinson *et al.* make this difficulty evident:

> MDE involves dependent activities that have both a positive and a negative effect. For example, code generation in MDE appears, at first glance, to have a positive effect on productivity. But the extra effort required to develop the models that make code generation possible, along with the possible need to make manual modifications, would appear to have a negative effect on productivity. [HRW11]

What is perhaps more fruitful is to focus on the specific "barriers to entry" and "lessons learned" in this case study. Barriers to entry denote aspects that are stopping unorthodox practitioners from engaging with the wither MDE practices. Lessons learned are specific to this project, but can be generalised somewhat to similar cases.

8.1 Lessons Learned

The lessons learned from this case-study are centred around code generation. The following three factors stand out:

- **Large undertaking**: before deciding to write a code generator, or to extend an existing one, it is important to understand the magnitude of the task and consider carefully if the organisation should be developing a code generator or using an existing tool, either proprietary or open source.
- **Well-defined purpose and scope**: the role of code-generation must be understood and agreed from the start of the project. Its better to have small, separate and well-defined roles and perhaps rely on several "special purpose" code-generators than to try a "one size fits all" approach, will is much harder to execute.
- **Separate prototyping from production**: code generation may be helpful in prototyping and exploring a problem space, but it may not be as suitable for the final system implementation. By having this in mind, development can proceed in stages: as parts of the system become understood, they can be moved to a final, better designed implementation.

8.2 Barriers to Entry

From the perspective of barriers to entry, three factors stand out of the empiric analysis:

- **Theory**: MDD / MDE are not widely known and understood. At the same time, MDD / MDE have clear practical application since software engineers are rediscovering significant parts of the methodology and applying it to the concrete problems they face. The MDE cannon appears to be too large and complex for an untrained developer to pick up without specialised training; yet, it is extremely difficult for a team without specialised skills to understand how to deploy it for their particular use case.
- **Tooling**: popular freely available tooling is connected to a specific IDE and/or platform — *e.g.* EMF — making it less appealing to those not using those technologies. It also appears to be tailored to advanced users such as researches and MDE practitioners rather than targeting entry level use cases. Developers cannot immediately see how to make use of the tools for their use case.
- **Developer Pipeline**: unorthodox practitioners can be thought of as proto-practitioners on the path to MDE, but which seem to struggle to find their way. There appears to be a *pipeline problem*, discouraging inexperienced practitioners to continue to develop their modeling skills towards the literature's state of the art. In this light, studying unorthodox practitioners is likely an important avenue in understanding what is blocking wider adoption of MDE.

Our conclusion, which points to future work, is that it seems there is a need for entry level tooling, able to integrate seamlessly with the most popular development environments — but not demanding

a specific one — and designed specifically to introduce inexperienced developers to MDE. The tooling should require little understanding of the theoretical apparatus, but simultaneously allow developers to explore more advanced topics as they become proficient in MDE. In addition, quantitative and qualitative studies should be performed on teams of unorthodox practitioners to understand what the requirements for the tooling should be. Finally, work should be carried out in the spirit evoked by Whittle *et al.* in [WHR14]: "match tools to people, not the other way around."

Acknowledgements The author would like to thank those that took part on the interviews and in particular, those who have facilitated the interviews and access to The Financial Company.

References

[Ame] David Ameller. "SAD: Systematic Architecture Design". PhD thesis. Universitat Politècnica de Catalunya (cit. on p. 1).

[And+14] Luigi Andolfato et al. "Experiences in Applying Model Driven Engineering to the Telescope and Instrument Control System Domain". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2014, pp. 403–419 (cit. on pp. 1, 2).

[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. 1, 6).

[Bec+01] Kent Beck et al. "Manifesto for agile software development". In: (2001) (cit. on p. 4).

[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 p. 8).

[FT00] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, USA, 2000 (cit. on p. 4).

[Fie+99] Roy Fielding et al. *Hypertext transfer protocol–HTTP/1.1*. Tech. rep. 1999 (cit. on p. 4).

[Fje08] Hans-Christian Fjeldberg. "Polyglot programming". MA thesis. Norwegian University of Science and Technology, Trondheim/Norway, 2008 (cit. on p. 4).

[HRW11] John Hutchinson, Mark Rouncefield, and Jon Whittle. "Model-driven engineering practices in industry". In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 633–642 (cit. on pp. 2, 13, 15).

[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 pp. 1–3).

[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. 7).

[MD08] Parastoo Mohagheghi and Vegard Dehlen. "Where is the proof? A review of experiences from applying MDE in industry". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2008, pp. 432–443 (cit. on pp. 1, 2).

[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. 1, 2).

[PL03] Randall Perrey and Mark Lycett. "Service-oriented architecture". In: *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*. IEEE. 2003, pp. 116–119 (cit. on p. 4).

[Pie06] Michael Piefel. "A common metamodel for code generation". In: (2006) (cit. on p. 6).

[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 p. 1).

[SKSG07] Dov Shirtz, Michael Kazakov, and Yael Shaham-Gafni. "Adopting model driven development in a large financial organization". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2007, pp. 172–183 (cit. on pp. 1, 3).

[Völ09] Markus Völter. "MD* Best Practices". In: *Journal of Object Technology* 8 (2009), pp. 79–102 (cit. on p. 1).

[Völ+13] Markus Völter et al. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013 (cit. on p. 1).

[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. 1, 16).

[Whi+13] Jon Whittle et al. "Industrial adoption of model-driven engineering: Are the tools really the problem?" In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2013, pp. 1–17 (cit. on p. 2).

[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 pp. 2, 4, 11).