Model-Driven Development Challenges and Solutions
Experiences with Domain-Specific Modelling in Industry

Juha-Pekka Tolvanen and Steven Kelly
MetaCase, Ylistönmäentie 31, FI–40500 Jyväskylä, Finland
{jpt, stevek}@metacase.com

Keywords: Model-Driven Development, Domain-Specific Modelling, Code Generators, Model Transformations, Language Workbenches, Modelling Tools.

Abstract: Model-Driven Development is reported to succeed the best when modelling is based on domain-specific languages. Despite significant benefits MDD has not been applied as widely as expected. Costly definition of languages and related generators with tooling, their maintenance when the domain is not stable, challenges in scalability, and collaboration are some reasons that several studies mention. We believe these statements are justifiable but only when applying traditional programming tooling for modelling. Instead we show with data from practice that many of the challenges reported can be solved when using tools built for modelling in the first place.

1 INTRODUCTION

Model-Driven Development (MDD), using models as the primary source when creating applications, is at a watershed. While many create models for communicating ideas and sketching solutions, the use of models to generate code divides practitioners. This is also visible in empirical research. For example, Petre (2014) states that UML-based MDD plays only a small role whereas Whittle et al. (2014) indicate that use of model-driven engineering is widespread. Clearly more empirical research on the use of modelling is needed.

Part of the reason for the different data is looking in different places. In our experience, MDD is less applied in project-based development, such as consulting, outsourced work, or IT as an internal support function. In contrast, MDD is common in product development, particularly in industries like automotive, telecom or banking. Also in areas like embedded architectures, testing, product lines or safety related embedded products, models play a major role. Some safety standards even expect a model-driven approach.

We focus on modelling that is based on domain-specific languages. Although we have acted as providers of both General-Purpose Languages (structured, object-oriented) and Domain-Specific Modelling (DSM), experience has shown us that DSM enables better MDD than general-purpose modelling languages. This is in line with recent studies like Whittle et al. (2014), who state: "The companies who successfully applied model-driven engineering largely did so by creating or using languages specifically developed for their domain, rather than using general-purpose languages such as UML".

The benefits of DSM do not come for free, as the language abstractions and tools to automate development need to be first developed and later maintained. Research claims that it is costly and hard to define modelling languages with tool support; that domain-specific languages can be created effectively only when the domain does not change; and that MDD does not scale.

In this paper we present our experiences, partly reported in cases over the last 25 years, first as researchers and then in industry. Unsurprisingly, most of our experience is with MetaEdit+. This however proves interesting, as our experience with a different tool indicates that some claims on MDD challenges are not true for all tools. We feel that one reason for our differing experience may be that the tooling we have applied is not what is traditionally expected — file-based tools built on top of programming IDEs — but instead developed specifically for DSML creation, and natively using a repository rather than XML or text files.

We start by looking at the evidence for MDD productivity gains and then discuss the cost factor:

Tolvanen, J-P. and Kelly, S.
Model-Driven Development Challenges and Solutions - Experiences with Domain-Specific Modelling in Industry.
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 711-719
ISBN: 978-989-758-168-7

how much it takes to create industrial-strength DSM languages, and if the cost varies with different tools. This is then followed by analyzing the language creation principles and supporting tooling: how languages are defined collaboratively to gain acceptance and better quality languages. We then look at the scalability of the use of the language in terms of large models and teams. Finally, we inspect how tooling and practices support the ongoing evolution of the DSM solutions, so that the gains in productivity can be maintained.

2 PRODUCTIVITY INCREASES

Survey evidence has shown that not all MDD approaches are alike. Here we will briefly consider quantitative empirical evidence for three approaches: UML, MDA, and DSM.

The evidence against a significant productivity increase with UML is unequivocal. Estimates of the effect on productivity of adding UML to coding vary between –15% and +10%. Djidek et al. (2008) is one of the more realistic studies, using professional developers and reasonably large non-greenfield tasks. Although some calculations in the study leave something to be desired, the data was clear: developing with UML and Java was 15% slower than pure Java (Figure 1).

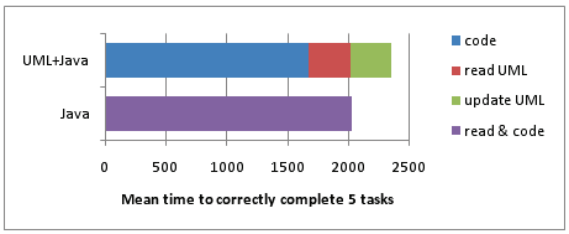

Figure 1: UML does not significantly improve developer productivity.

Another candidate for an MDD approach is the OMG's MDA. Tool vendors' own figures for MDA's productivity increase include +22% (Obeo, 2014) and +30% to +40% (OptimalJ, 2003). These figures are based on code-generation approaches with largely UML-based MDA tools. While better than plain UML, these still do not represent the paradigm shifting magnitude that the industry is looking to obtain from MDD.

Domain-Specific Modelling predates UML, but widespread use only began in the latter years of the last century. Although any particular DSM language will, by definition, have limited applicability, the approach itself seems to suit a wide range of domains. The literature provides numerous DSM cases from various industries, such as industrial automation, government acquisitions, automotive, avionics, command and control systems, robotics, secure networks, education, medical treatment, and autonomous-vehicle development (Sprinkle et al., 2009).

Although there are a number of DSM tools, to our knowledge only MetaEdit+ has accumulated quantitative evidence from a number of empirical evaluations and experiments. Part of the explanation may be its long history, wide use, and the research background of the principals. Nokia (MetaCase, 2000), Panasonic (Safa, 2007), Polar (Kärnä et al., 2009), Elektrobit (Puolitaival et al., 2011) and Ouman (Puolitaival, 2011) all report significant productivity improvements using DSM languages in MetaEdit+ (Figure 2). The increased productivity on tasks now handled by DSM ranged from 400% to 2000%, with most being in the range of 500–1000%.

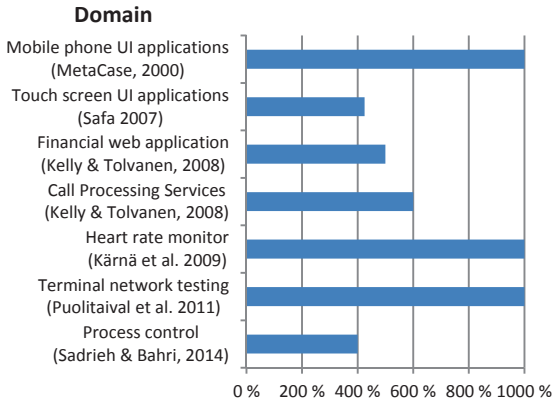

Figure 2: Productivity gains reported with DSM.

This "5–10x productivity" result has taken on something of a life of its own: while gratifyingly consistent among MetaEdit+ users, it is also widely quoted by others in support of their own DSM tools, and sometimes even for completely different MDD approaches. Given the absence of evidence for such broader applicability, there is clearly a need for further empirical research with other tools, and for investigation of which elements in these MetaEdit+ cases contributed to the productivity increase. Cases handled entirely by customers have been able to rule out the possibility that the 5–10x is restricted to languages made or guided by MetaCase consultants.

There remains the possibility that MetaEdit+ itself is a factor. If so, it seems plausible that the effect would be more from the language definition phase than the language use phase. One explanation

could be that a tool that makes language development easier allows developers to concentrate on making a better language.

3 DSL CREATION EFFORT

Creating DSLs is stated to be hard (Mernik et al. 2005) and to require time and resources – in particular when creation of tooling support is included (Mohagheghi et al. 2013). Unfortunately the vast majority of the reports on modelling language creation do not disclose much about the effort. In conference talks some figures are given, like 25 man-years for creating a commercial UML tool (Ströbele, 2005), 3.5 man-years for creating tooling for insurance product modelling (Warmer & Bast, 2011), or 35 persons working on creating modelling tools within a single company (Bordeleau, 2014). Few provide more detailed figures, or break development effort down into different parts like language, generators, tools etc.

Reasons for this lack of quantitative evidence may include that companies implementing their tooling do not want to share such figures, or that they are not systematically collecting data on resource use. Languages created by academics are often not defined completely as they were not intended to be deployed in practice: the focus was on studying particular language features, tools or ideas rather than making a full DSM solution.

Figure 3 illustrates DSM development effort for various domains in industrial cases in which the authors have been able to have access to the development process. The actual language and generator development with MetaEdit+, however, has been done by the customer themselves, as has the measurement of time taken.

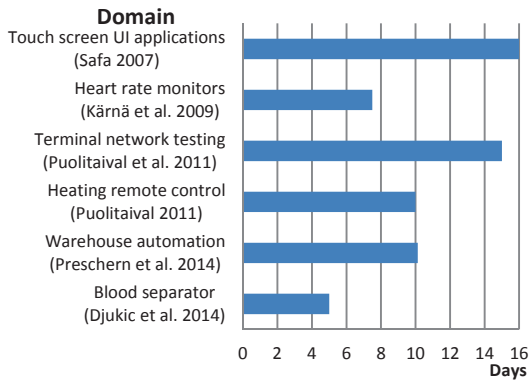

Figure 3: Days to define modelling languages and code generators along with tooling support in MetaEdit+.

These cases show that the effort for companies to create their own DSLs and tool support need not be as costly and time-consuming as many studies claim. Our experience in other cases confirms that times of 5–15 man-days are normal with MetaEdit+.

Comparison of development effort remains hard because the domains the DSM solutions address vary for example in size, complexity, or detail; the expertise of the development team can vary widely; and different languages and tools are used to create DSM solutions. These factors are hard to compare in real-world empirical research – it is hard to find cases in which multiple DSM solutions are created for the same domain with comparable development teams and the same tools. Comparison across domains seems fruitless; comparison of low and high experience teams would presumably be obvious; comparison between tools seems useful.

An empirical study by El Kouhen et al. (2012) indicates that tools have a large effect on the amount of effort required to develop a language (Figure 4). Implementing the same modelling language took 50 times longer with the slowest tool (GMF) than with the fastest (MetaEdit+), and even the second fastest tool was 10 times slower than the fastest. Interestingly, this was despite the participants being most familiar with Eclipse EMF / Ecore, on which these slower tools and RSA were based.

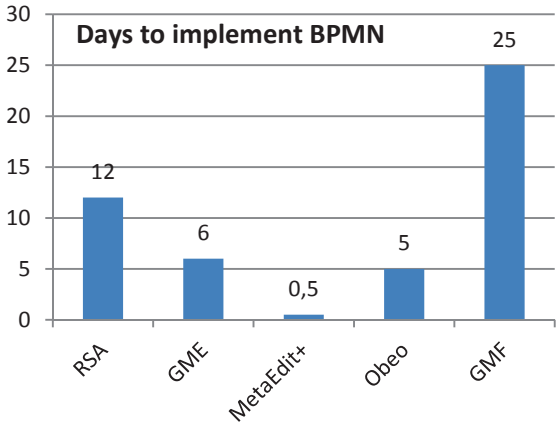

Figure 4: Effort to define the same BPMN language with different tools.

The results were also in stark contrast to the Eclipse researchers' assessment of the five tools before measurement. They graded each tool with a subjectively assessed "efficiency" score, and the tool with the worst "efficiency" turned out to be the fastest, while the second best score turned out to be the slowest. Similarly, in "task visibility" (a sub-category of "usability"), the worst score was given

again to the tool that turned out fastest, and the best score to the slowest. It seems that factors considered by Eclipse researchers to contribute to efficiency and usability may actually be anti-patterns. Compounded with the lack of quantitative empirical research by the Eclipse community on DSM solution creation times, this belief in anti-patterns could become self-sustaining. In that sense it is a shame that the article in question has not been published outside of its organizational repository.

A possible explanation for the difference in speed with different tools could be found by applying DSM to itself. Unlike UML and its subset used in Ecore, MetaEdit+'s GOPPRR is a domain-specific language, designed from the ground up for describing modelling languages. Kern et al. (2011) compared six metamodelling languages, concluding that in terms of power and expressiveness GOPPRR > GME > DSL Tools > Ecore: the same order as the productivity of the respective metamodelling tools (taking the average of the three Ecore-based tools).

Another explanatory factor could be a good match of language to tool. BPMN has a relatively large number of graphical symbols, and MetaEdit+ was the only tool with a WYSIWYG graphical symbol editor.

In addition to comparing tools, one can compare approaches for defining languages, e.g. UML profiles vs. metamodelling. In a case of railway DSL, the size of static semantic rules with UML profile and OCL was two times larger than when been defined in native metamodelling language GOPPRR (Mewes, 2009). While the length of the definition (LOC) does not describe the effort for creation and maintenance, it gives one figure to estimate the effort.

We look forward to other case studies and language creation reports indicating some data on the development effort, size of the team, time used etc. Establishing good metrics for collecting and analysing the effort is challenging, particularly for industrial cases. Ideally research would collect data for different elements or phases of language creation (abstract syntax, concrete syntax, static semantics, tooling, etc.). However, even obtaining a single figure is hard enough. For instance, Polar initially planned to continue gathering empirical data of the productivity increases, but soon stopped, as the evidence of return on investment was so clear that time spent investigating further was no longer commercially justified. One potential source of good empirical data with low additional effort could be 'language workbench challenges' (Kelly, 2013). Some encouraging preliminary results have already been achieved in this manner (Erdweg et al., 2013).

4 COLLABORATIVE LANGUAGE ENGINEERING

The more esoteric the skills required to create a language, the further that creation will be from the people who will actually use the language. By making language creation easier, a tool can democratize the process of developing a DSM solution. In our experience with MetaEdit+, language engineers and language users are often at least partly the same persons – in particular if the domain addressed is small or within a single company.

Another characteristic of industrial-scale use is that a single language is often not enough: there must be multiple integrated languages, which requires collaboration among language engineers too.

4.1 End-user Participation

Only a few tools have been implemented to support collaborative language engineering (Rossi et al. 2004). These have focused on capturing and discussing design rational related to the metamodel (Oinas-Kukkonen, 1996) and emphasizing the role of the end users in the language design process (Izquierdo et al., 2013). While these features are promising little has been reported on their use. Instead it is common that language definition part and language use part are separated into different tools. This separation makes then hard, if not impossible, collaboration during language engineering and maintenance. Even worst case is tools where the language definition is isolated into a single file that one person can edit at a time. This language definition is often then presented in a format that is not easy to follow by language users – if they can even access it. It also prevents any reflection from language use back to the language definition. For example to trace which parts of the language, its elements and constraints, are been used, which have been hard to use, not used, or users have most difficulties to apply.

Since active participation and feedback from users is also crucial during language engineering a good way is to add such capabilities to the language itself. Language may include concepts dedicated directly to extract experiences, collect feedback, or get requirements for improvements. For example, to

collect feedback from language users, in one case at Panasonic a special 'Joker' concept was added to the language to collect feedback from language users (Safa, 2007). In another case, while implementing automotive architecture DSL several persons and organizations have been involved (Tolvanen et al., 2014) the collaboration and feedback is gathered by adding constructs for commenting and gathering feedback to the language itself to be used by both language engineers and language users. One main benefit this "language-based" approach provides is that the feedback is tightly related to the language use and to the language specification.

Third and based on our experiences the most powerful way for collaboration is using the language while it is defined at the same time - agile in its extreme. For example, with MetaEdit+ tool the language can be defined and used at the same time even in the same tool. This is not restricted to single person only as with MetaEdit+ multi-user version we have been working in cases where language is formulated in group sessions: ten persons have been using the newly provided language and provided feedback on its capabilities. In parallel changes are then implemented to the language and then immediately tried out. This has provided several benefits: errors in language definition are quickly identified, changes can be made instantly, and ideas can be demonstrated with concrete examples on the model level. For most of us it is easier to see how the language works in practice than comment the metamodel. This approach brings also similar benefits typical to participatory approaches: language will have better quality, it improves its acceptance, and it organizational introduction is easier.

4.2 Collaboration within the Language Engineering Teams

In industrial use companies usually have several language engineers. On the one hand the responsibility is often wanted to be shared to share dependencies on few persons. On the other hand, when several languages are used it is not even realistic to expect that a single person handles them all. In our experience it is not unusual even with a single language to divide the work into different parts for different people: one defines the metamodel, another the generators, and a third tests them. It is therefore natural to expect that language engineering processes and tools would support collaboration within the team.

Most modelling tools, however, focus on a single language at a time. On metamodelling level, languages like MOF do not even have a concept of language so integrating then several ones is challenging. Tool platforms that are built on the basis of multiple languages and their language integration, like MetaEdit+, MPS and Spoofax, solve the integration already as in-built characteristic of the tool (Cheng et al., 2015). MetaEdit+ for instance enables several language engineers working on the same language definition at the same time. Access rights can be given to user accounts and security level can be given to restrict the number of language engineers per repository, per project there or per individual type.

For example in the case of automotive embedded systems, a language called EAST-ADL includes over 20 different sub-languages each covering different subdomains (architecture, safety, error modelling, requirements, variability, hardware etc.). A single person can hardly master them all along. Instead the definition of the metamodel part was initially divided among three language engineers – each focusing on different sub-languages. Integration of these languages then becomes easy as common types among the languages form the "glue" to integrate them. Yet this team was accompanied by other language engineers focusing on implementing generators for various targets. This speeded the implementation, allows checking and verifying the language definition early and testing the whole in collaboration.

5 SCALABILITY

For several years, the MDD tooling in focus in the majority of academic research articles has been Eclipse EMF. This is in stark contrast with the lack of articles reporting industrial-scale use of graphical DSLs in EMF, in particular quantitative empirical comparisons of scalability or productivity for language use or creation in industry.

Research articles often cite scalability (e.g. Gómez et al., 2015, Pagán & Molina, 2014) as an area of EMF that needs improving before widespread industrial adoption could be possible. There are many articles focusing on such improvements: Kolovos et al. (2013) provides an overview. The scalability improvements can be divided into two main areas: handling large models with reasonable performance, and supporting multiple simultaneous modellers working on the same models.

Rather than dwell on the challenges of other

tools, we shall briefly present here the ways in which the architecture, design and implementation of MetaEdit+ address scalability for large models and large teams. As MetaEdit+ does not use the XML files and diff+merge common in EMF, seeing its different approach and results may be useful to other tools.

5.1 Large Models

MetaEdit+ has been used industrially since 1995, scaling to cases lasting decades with hundreds of users and gigabytes of models. An object repository is used to store both metamodels and models, including both conceptual and representational (abstract and concrete syntax) data for both. A repository can consist of an unlimited number of projects, each of which can hold over 4 billion persistent objects. Models and metamodels can be divided across projects, and reference others across projects, as the users desire. The repository can be used on a user's hard disk in single user mode, or from a server in multi-user mode.

When logging in, MetaEdit+ pre-loads an initial subset of the repository based on the metamodel structure. As further links are followed by the user opening models, the necessary objects are loaded. If memory is short (based on configurable parameters), a portion of the objects are automatically flushed from memory to make room to load others, allowing work to proceed through more objects than would fit into memory at one time.

As loaded objects are directly the objects of the model, with no intermediate proxies, XML representations, or other overhead, working with them is equally fast for both large and small models. Loading and saving read and write objects in binary format on disk or over the network, but only needed objects are read, and only changed objects are written.

During generation, only changed output files are written, allowing build tools to compile only those (compilation is far slower than MetaEdit+ generation). This is completely automatic: there is no need for generator developers to isolate generators to limit a single output file to a single input graph. Instead, MetaEdit+ caches generator results and compares with output files already existing on disk, only writing those files that have changes. A generator can thus freely access any information in any graph, and produce any part of any output file. This allows the language to be made to present the most relevant information together in a graph, regardless of the requirements for information distribution across the output files.

Together, these factors allow MetaEdit+ to work with larger files faster than any other graphical DSM tool. At the Language Workbench Challenge in 2014, the aim was to open a model with 2^{10} objects. (These were main objects with some further details inside: similar to a Class or State in a UML model.) MetaEdit+ demonstrated a repository with 2^{20} objects, taking up over 5 GB on disk. Opening a project of 2^7 graphs, each with 2^7 objects, took under a second. Generating a full application of 360kB of source code from a graph of 2^{10} objects took 2.1 seconds. At that size, well beyond the common size of 30–40 objects for a DSM graph, generation has become $O(N^2)$: although MetaEdit+ was the fastest of the graphical tools, there is always room for improvement.

5.2 Large Teams

A company can have a single MetaEdit+ repository or split their work into multiple repositories as they desire, e.g. one per team, one per user, or one per module. Where repositories are to be used by more than one person, concurrent access is handled by the MetaEdit+ multi-user server. Each persistent object may have many simultaneous readers but only one writer; the granularity of such locking is fine, down to the level of a single property of an object. This is in clear contrast to the approach of EMF, where the level of granularity of simultaneous editing is the XML file – a model or set of models. (Add-ons such as CDO do not yet substantially change that picture: as Kolovos et al. (2013) point out, CDO offers "no mature support for conflict management and merging," "does not scale up as well as advertised," and "failed to load all test sets greater than 271MB.")

By using a multi-user object repository, MetaEdit+ is able to avoid the need for users to make their own copy of a model while they work on it, and to have to merge those changes back together. The user can still choose when to release his changes to others: until that point, they will see the latest released version.

These long-lived ACID transactions and fine-granularity locking have been found to work well for design work. It appears that database usage in design work is unlike normal database applications in a number of respects (Welke, 1988). For instance, a common database solution to a collection that will be modified by many users is a B-tree. However, that is fundamentally unsuited to the collection of graphs held in a project. That collection will have

the most simultaneous changes at the start of a project, when it is smallest and thus a B-tree's support for multiple modifiers would be at its worst. This called for the creation of a novel kind of multi-user data structure (Kelly, 1998), which has proven to work well without a need for manual tuning. For those who are interested, further details of the repository implementation are available in that article and in the MetaEdit+ manuals (MetaCase, 2014).

6 MAINTENANCE AND LANGUAGE EVOLUTION

If the domain stays stable the language can be defined once and frozen. This does not happen in practice as domains always evolve, language users learn new and better ways to specify systems that the modelling language should support, and the initially created languages are recognized to have some unwanted features (Kelly & Pohjonen, 2009). Perhaps more so than with GPL, a DSL must evolve, and thus the models already created with it need to evolve correspondingly.

So far the majority of research and tools have focused on the initial phase of language creation, rather than on the maintenance of the language and the models made with it. For example, even the most broad-coverage studies focusing on tools (Erdweg et al, 2013; El Kouhen et al., 2012) do not address maintenance. This is somewhat surprising given the accepted software development wisdom that maintenance is a far larger effort than the creation of the first release.

In on our experience, the changes for any well-piloted language are typically not fundamental ones that change the nature of the language. Instead, maintenance tasks involve changing some particular language elements, renaming parts, adding new ones and removing or sunsetting others. In the worst case, when the domain the company addresses with the products changes completely, the update of the DSL part is not the major issue: it is more likely they will consider creating a totally new language (see Section 3).

Unfortunately, in many tools even small changes are fatal for the tooling. The authors are aware of two automotive cases in which Eclipse-based tools were used to define modelling editors for AUTOSAR (an automotive related metamodel). In both cases these tools were abandoned as work done in previous AUTOSAR modeling editors could not be opened with a newer AUTOSAR version. While both of these cases were in the Eclipse Modelling Platform, similar experiences are also reported in the DSL tool developed on top of Visual Studio. Adding Language Workbench capabilities on top of a programming IDE seems to be hard. Perhaps this is because IDE users are not used to expecting better from support for traditional textual languages, where changes to the language often require manual updates to source code, or character-based find/replace tools (France et al. 2013).

Industrial usage reveals unwanted practices quickly. In MetaEdit, the predecessor of MetaEdit+, changes to the metamodel often prevented opening models from earlier versions: a commercially untenable situation. It was then decided to analyse language evolution cases and build automated support for updating models to follow metamodel changes. While this functionality is described (MetaCase 2014) and available to download, we give a few examples of typical maintenance tasks that we have seen to occur in practice:

- Renaming of a language element is automatically reflected to existing models: they follow the new name. Also renaming in the abstract syntax is automatically updated to the concrete syntax and static semantics.
- Changed constraints are followed automatically and shown for the language user when opening them in the editors.

These model migrations to the new metamodel happen automatically without the language developer needing to do any additional work. The hardest parts are then the changes that require human intervention based on the metamodel changes. For this kind of language refinement situations the language engineer can inspect the existing models to see possible consequences of the metamodel update. This is important as typically a change to one element in the metamodel has an influence on other concepts.

To support human migration of the models the language engineer can make checking reports and model annotations that show which elements require update. This way after each language version release, language users can easily see which parts of the model need to be updated along with possible guidelines based on the new metamodel for doing so.

One indication of the viability of the approach taken in MetaEdit+ is that we are aware of customers today using DSM languages and generators which have been updated in a rapidly changing domain and were originally developed in

the mid 90's: 20 years of DSL evolution. A good topic for future research would be inspecting DSM/DSL evolution cases with various tools, languages and domains to identify which maintenance approaches work better than others.

7 CONCLUSIONS

Two main claims are made around MDD: firstly, that it can increase productivity to 500–1000%, and secondly that creating a DSM language is costly and difficult. Our research indicates that these claims are true, but for disjoint data sets:

- UML-based MDD or MDA offers productivity improvements of only up to 40%; 500%–1000% productivity has only consistently been reported in cases applying DSM in MetaEdit+.
- Creating an industrial-scale DSM language, generators and editor consistently takes 5–15 days in MetaEdit+; independent experimental evidence indicates that implementing the same language in any other current tool takes 10–50 times longer, fitting the commonly reported time of several months with Eclipse tooling.

These are strong statements, but backed up by strong empirical evidence. We believe that there is no reason why similar 500–1000% productivity increases could not be achieved in other modelling tools, if they were to implement the same DSM languages as in MetaEdit+. Rather, the reason for poor productivity in the resulting languages is that the languages differ, with those in the other tools being dragged down by the difficulty of language creation in those tools. Reducing that difficulty appears hard, at least if building on top of Eclipse EMF and GMF.

Creation of DSM solutions that are productive in use and address user needs is easier if languages and generators can be created in close collaboration with language users, and without having to think about low-level details of the tooling implementation. MetaEdit+ keeps language definition on a high level of abstraction, and lets language users use the modeling language at the same time as it is being refined. Its ability to allow changes in the language and reflect them automatically in existing models is useful during both language creation and evolution. Similarly, the scalability challenges of large models and multiple concurrent modelers are easier to address when models are not handled as text or XML files but with a native multi-user repository.

We believe that the way forward is for researchers to widen their scope beyond Eclipse, despite its obvious attractions. Empirical researchers should look to take today's most effective tools into industrial settings, and tool developers should identify the differences between tools and approaches that explain the order of magnitude differences in results.

REFERENCES

Bordeleau, F., 2104. Papyrus and Open Source Modeling -- Status, Strategy, and Plan. Presentation at Ericsson Modeling Days, 4 November 2014, Kista, Sweden.

Cheng, B., Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., (eds), 2015. *Globalizing Domain-Specific Languages*, Springer, LNCS 9400.

Djukić, V., Popović, A., Tolvanen, J.-P. 2014. Using domain-specific modeling languages for medical device development, *Embedded.com.*

Dzidek, W.J., Arisholm, E., Briand, L.C., 2008. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance, IEEE Transactions on Software Engineering, Vol 34 No 3, May/June 2008.

El Kouhen, A., Dumoulin, C., Gérard, S., and Boulet, P., 2012. *Evaluation of Modelling Tools Adaptation*. CNRS HAL. http://hal.archives-ouvertes.fr/docs/00/70/68/41/PDF/Evaluation_of_Modelling_Tools_Adaptation.pdf.

Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J. 2013. The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge. *Software Language Engineering*, LNCS 8225, Springer, pp 197-217.

France, R., Rumpe, B., Schindler, M. 2013. Why it is so hard to use models in software development: observation. *Software & Systems Modeling*, Volume 12, Issue 4, pp. 665-668.

Gómez, A., Tisi, M., Sunyé, G., Cabot, J. (2015) Map-Based Transparent Persistence for Very Large Models. *Fundamental Approaches to Software Engineering*, Springer, pp 19-34.

Izquierdo, J. L. C., Cabot, J., López-Fernández, J. J., Cuadrado, J. S., Guerra, E., & de Lara, J. 2013. Engaging end-users in the collaborative development of domain-specific modelling languages. *Cooperative Design, Visualization, and Engineering* (pp. 101-110). Springer Berlin Heidelberg.

Kelly, S. 1998. CASE Tool Support for Co-operative Work in Information System Design. In Proceedings of the IFIP TC8/WG8.1 Working Conference on Information Systems in the WWW (pp. 49-69).

Kelly, S., 2013. Empirical Comparison of Language Workbenches. In Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling (pp. 33–38).

Kelly, S., Tolvanen, J.-P., 2008. Domain-Specific Modeling: Enabling Full Code Generation. Wiley.

Kern, H., Kühne, S., Hummel, A., 2011. Towards a comparative analysis of meta-metamodels. In DSM'11, Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, VMIL'11, SPLASH '11 workshops, ACM.

Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi., M., Cabot, J. 2013. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In Proceedings of the Workshop on Scalability in Model Driven Engineering, ACM.

Kärnä, J., Tolvanen, J.-P, Kelly, S. 2009. Evaluating the use of domain-specific modeling in practice. *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*.

Mernik, M., Heering, J., Sloane, A., 2005. When and How to Develop Domain-Specific Languages, *ACM Computing Surveys*, Vol. 37, No. 4, pp. 316–344.

Mewes, K., 2009. *Domain-specific Modelling of Railway Control Systems with Integrated Verification and Validation*, dissertation, University of Bremen.

MetaCase, 2000. Case Study: MetaEdit+ Revolutionized the Way Nokia Develops Mobile Phone Software. www.metacase.com/papers/MetaEdit_in_Nokia.pdf.

MetaCase, 2014. MetaEdit+ 5.1 Manuals. http://www.metacase.com/support/51/manuals/

Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M., Nordmoen, B., Fritzsche, M. 2011. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling*, Volume 12, Issue 3, pp. 619-639.

Obeo, 2014. "18%: The average cost reduction of a project with the MDA approach". web.archive.org/web/2014 0209081524/http://www.obeo.fr/pages/presentation/en.

Oinas-Kukkonen, H. 1996. Method rationale in method engineering and use. *Method Engineering* (pp. 87-93). Chapman & Hall.

OptimalJ, 2003. www.theserverside.com/news/1377283/ TheServerSide-Symposium-June-2003-Coverage.

Pagán, J. E., Molina, J. G. 2014. Querying large models efficiently. Information and Soft-ware Technology 56(6), pp. 586–622.

Petre, M. 2014. "No shit" or "Oh, shit!": responses to observations on the use of UML in professional practice, *Software & Systems Modeling*, Volume 13, Issue 4, pp. 1225-1235.

Preschern, C., Kajtazovic, N., Kreiner, C. (2014). Evaluation of Domain Modeling Decisions for two identical Domain Specific Languages. International Conference on Software Technology and Engineering, Lecture Notes on Software Engineering (LNSE), Vol. 2, No.1.

Puolitaival, O.-P., 2011. Home automation DSL case, *Presentation at Code Generation Conference* (http://codegeneration.net/cg2011/).

Puolitaival, O.-P., Kanstrén, T., Rytky, V.-M, Saarela, A. (2011) Utilizing Domain-Specific Modelling for Software Testing, *The 3rd International Conference on Advances in System Testing and Validation Lifecycle,* October 23-29, 2011, Barcelona, Spain.

Rossi, M., Ramesh, B., Lyytinen, K., & Tolvanen, J. P. 2004. Managing evolutionary method engineering by method rationale. *Journal of the Association for Information Systems*, 5(9), 12.

Sadrieh, A., Bahri, P. 2014. Novel Domain-Specific Language Framework for Controllability Analysis, *Computer Aided Chemical Engineering*, Volume 33, pp. 559–564.

Sadrieh, A., Bahri, P., 2014. Novel Domain-Specific Language Framework for Controllability Analysis. Computer Aided Chemical Engineering, Elsevier, Volume 33, pp. 559-564.

Keywords: Domain-Specific Language; Maintainability; Controllability analysis; CAPE.

Safa, L. 2007. The making of user-interface designer a proprietary DSM tool. In *7th OOPSLA workshop on domain-specific modelling* (DSM).

Sprinkle, J., Mernik, M., Tolvanen, J-P., Spinellis, D., 2009. What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software*, July/Aug.

Ströbele, T., 2005. EclipseUML—UMLundEclipse, presentation at OOP Conference, 24–28 January, 2005, Munich, Germany.

Tolvanen, J.-P., Luoma, J., Chen, D., -J. 2014. Reaping the benefits of architectural modelling in embedded design. *Embedded*, November.

Warmer, J., Bast, W. 2011. Developing an Insurance Product Modeling Workbench. Presentation at Code Generation Conference 2001, Cambridge, UK.

Welke, R.J., 1998. The CASE Repository: More than another database application. In Proceedings of 1988 INTEC Symposium Systems Analysis and Design: A Research Strategy, Atlanta, Georgia, Cotterman, W.W. and J.A. Senn (eds.), Georgia State University.

Whittle, J.; Hutchinson, J.; Rouncefield, M., 2014. The State of Practice in Model-Driven Engineering, *IEEE Software*, vol.31, no.3, pp.79-85.