Effort Used to Create Domain-Specific Modeling Languages

Juha-Pekka Tolvanen
MetaCase
Jyväskylä, Finland
jpt@metacase.com

Steven Kelly
MetaCase
Jyväskylä, Finland
stevek@metacase.com

ABSTRACT

Domain-specific modeling languages and generators have been shown to significantly improve the productivity and quality of system and software development. These benefits are typically reported without explaining the size of the initial investment in creating the languages, generators and related tooling. We compare the investment needed across ten cases, in two different ways, focusing on the effort to develop a complete modeling solution for a particular domain with the MetaEdit+ tool. Firstly, we use a case study research method to obtain detailed data on the development effort of implementing two realistically-sized domain-specific modeling solutions. Secondly, we review eight publicly available cases from various companies to obtain data from industry experiences with the same tool, and compare them with the results from our case studies. Both the case studies and the industry reports indicate that, for this tool, the investment required to create domain-specific modeling support is modest: ranging from 3 to 15 man-days with an average of 10 days.

CCS CONCEPTS

• **Software and its engineering** → *Domain specific languages*; *Model-driven software engineering*;

KEYWORDS

Domain-specific language, modeling, code generation, development cost

ACM Reference Format:

Juha-Pekka Tolvanen and Steven Kelly. 2018. Effort Used to Create Domain-Specific Modeling Languages. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239372.3239410

1 INTRODUCTION

Domain-Specific Languages (DSL) and Modeling (DSM) have been shown to significantly improve the productivity and quality of software development in various industries, such as automotive, consumer electronics, signal processing, telecom, solar power systems, military etc. [4, 15, 27, 32]. These improvements are possible because of two characteristics. First, a DSL can raise the level of abstraction beyond programming by specifying the solution in a language that directly uses concepts and rules from a specific problem domain. Second, generators can produce fully functional code from the high-level specifications [11, 27]. History has shown that each time the level of abstraction has been raised with automation, the improvements have been significant.

Typically the benefits from DSM or DSLs are reported without detailing the size of the original investment: How many people and how much time did it take to create the modeling languages, their generators, and tool support. In this study we address this omission by comparing the effort to create such complete DSM solutions across ten cases.

We investigate the size of the investment by analyzing language creation effort in two complementary ways: by conducting case studies ourselves, and by analyzing reported industry cases. We focus on languages created and used in real-world situations. These language implementations are complete and ready to be used. Using a case study research method we collect data on the implementation effort in detail in two cases, and with a literature review we inspect publications from eight industrial language creation projects that also reported the initial development effort. The results of the comparison across all cases show that with appropriate tools, the development effort is modest — in particular when compared to the benefits achieved.

After describing the research methods and subject of this study, we report the case studies and industry experiences, and compare the effort across them. We then conclude by examining other published comparisons of language development effort, and integrating our results into that body of evidence.

2 INVESTIGATING LANGUAGE CREATION EFFORT

Creating DSLs is said to be hard [15], and to require time and resources — in particular when creation of tooling support is included [19]. Unfortunately the vast majority of the studies on creating modeling solutions do not disclose much about the effort. Investigating the creation of DSLs is also challenging as it is hard to obtain data that is comparable and could be generalized. First, there is the obvious difference between the domains, making it hard to compare language creation projects. Second, there are differences in the experience of the language developers. Third, tools and technologies applied for language development influence the effort needed. In this study we exclude the influence of different tools by focusing on language creation efforts within the same tool, MetaEdit+ [9, 17].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239410

2.1 Subject of the Study: DSM Solution

By language creation we mean creating a complete modeling solution that can be given to language users. Typically the following parts need to be specified for a complete DSM solution:

Metamodel: Language creation almost always starts from identifying the language concepts. The bare concepts are fleshed out with their properties and relations to form a language. The resulting language is then formalized into a metamodel or a grammar in a tool.

Constraints: In addition to the language concepts, there are also a number of rules and constraints that models should follow. Defining them aids in prevention and early identification of modeling errors, speeds up specification work, and makes the models applicable for generators, checkers, simulation etc. These rules are typically part of the metamodel or defined separately with a language expressing constraints.

Notation gives a representation for humans to create, edit and read the specifications. Typically, a symbol is defined for each of the main concepts in the modeling language. Additionally there can be notational elements to show errors, incompleteness, provide reading aids etc. Often the notation is a diagrammatic representation, but can also be based on other representational paradigms like matrices or tables. In this study we focus on domain-specific modeling and exclude solely textual representations and extensions of programming languages — along with related syntax-oriented text editors.

Generators read the models and transform them into code or other outputs. Building a generator is about defining how model elements are mapped to code or other output. Generator building also includes making sure the resulting code is ideal: optimization, using patterns and coding styles, and integration with existing libraries and legacy code.

Tooling and tool-chains: Depending on the tool being used, in addition to defining the language's metamodel, constraints and notation, it may be necessary to provide extra information or code to implement modeling tool functionality. Generators too will depend on tooling, and need to be integrated with other tools in the chain: the compiler, build process, requirements management, version control, testing etc.

In addition, the language creator can make ancillary material over and above what is necessary to provide tool support for the language: training material, tutorial examples, guides etc. As these are similar to those for any other task or technology, and the set is rather open-ended, we will only consider them in passing here.

During the literature review we did not find any publications that detailed the implementation effort for all of the above parts. The selected case study research method allows us to do that.

2.2 Research Methods

To obtain a more comprehensive view on the language development effort we apply two research methods: case studies and review of industry reports.

2.2.1 Case Studies. With a case research method we collect detailed data on two DSM creation projects: one addressing creation of applications for an embedded device and another implementing an enterprise architecture language called ArchiMate [28]. These cases were selected as they are of realistic size and their implementation can be made available without any restrictions. These languages are also different in respect to their purpose, and in the way their existing specification is presented, particularly when addressing constraints and generators.

As in all research, a case study must be planned to obtain scientific knowledge, and to overcome or minimize the limitations of the research approach followed. Our research objective is to extract information in detail about the development effort for creating the different parts of the whole DSM solution.

During the case studies the language creation followed the typical iterative approach [11, 33], defining the language in small parts and testing the resulting definition by using the language with practical examples. The language implementation was performed by one engineer, as in most other industry cases (see Section 4), with some assistance from a language user testing it. The language engineer was the same in both cases, and was not familiar in detail with the domains but learned about them while implementing the DSM solutions. The tool used for defining and using the DSM solution was already familiar. Language support was tested by asking language users to apply it, by checking the result with reference applications and by using certification documents when available.

In both cases the modeling support was implemented completely, covering the metamodel, notation and semantics along with supporting tooling. Both resulting DSM solutions are ready to be used, along with integrated help and sample models.

During the implementation of the language support we collected data in two main ways: Keeping a manual record of working hours and using log data provided by the tool. The latter is more precise as it is collected automatically, but it did not provide data on all parts of the DSM solution: MetaEdit+ can provide log information when a particular language concept was created (timestamp in seconds), when the language definition effort started and when it was stopped by saving the work (times of transactions). Also the timestamps when generators were created or changed can be inspected within the tool.

In addition to the automatically collected logs, the language engineer can version the changes — covering the models and the metamodel. Both automatically collected and manually created versions were used to obtain a detailed view on the development times. MetaEdit+ does not automatically collect log information related to the notation and therefore the effort related to creating notations is based on manually kept records.

Others can repeat the case studies by collecting the data on the effort to implement languages following the same elements as we did. The scope of future studies can also be extended for example by implementing other languages or by using different tools.

2.2.2 Review of Industry Reports. In addition to case studies, we reviewed publicly available cases from various companies and domains. This allowed us to compare the findings from the case studies with industry reports and obtain a richer picture of the language creation effort. To avoid the authors' influence as individual language engineers, we excluded industry reports where the authors had been directly involved, whether more generally in the language creation process, or more concretely in actually using MetaEdit+ to implement the DSM solutions.

Investigation of industry experiences was harder than expected as companies usually do not report about their investments or effort even on a general level — e.g. only a few among the 50 cases at [6]. This is partly understandable as companies do not want to share data on their investments, or necessarily even to spend time collecting the data. Where data is provided, it may be expressed in a way that prevents identifying the effort used for creating the language with supporting tooling. For example, at Ericsson Modeling Days a talk [1] indicated that the company has worked for more than 5 years on introducing a particular modeling tool, or has 35 people working on creating modeling tools [3].

We focused on papers and presentations describing a particular DSM solution created within a company, where details about the development effort were provided. The reports were selected from search engine query results using domain-specific modeling or domain-specific and model as keywords, and focusing on reports which have a clear industry relationship. Therefore for example cases performed solely by researchers were excluded, even if published in industry tracks of conferences. We noticed that academic publications describing language creation tend to focus on the language itself, its use, or related tooling but seldom include information about the actual development effort. It is also common that the language is not necessarily defined completely (e.g. only a metamodel) as the focus has been on studying certain aspects rather than providing a complete and ready-to-use DSM solution. Since we focused on one tool to control for the influence of tools, which earlier research has shown can be substantial [10, 12, 13], the selected literature was restricted to MetaEdit+ based DSM solutions.

3 LANGUAGE DEVELOPMENT EFFORT IN TWO CASES

The language developers had access to all documents related to the domain and/or language. For the first case, addressing an Internet of Things (IoT) device, identifying the domain concepts was quite straightforward as the device had a fixed set of sensors and functionality — all detailed in the device documents. Putting these concepts together into a language was relatively simple: a domain-specific state model approach seemed appropriate. Notation, model checking rules and generators needed to be identified and developed. The code generator was tested with the available reference applications.

The implementation of the ArchiMate metamodel followed existing specifications by The Open Group and thus shows a case in which almost all requirements are available. Thus, it is similar to other studies made to compare the effort where a known language is defined (e.g. [13, 21]). Implementation of ArchiMate was tested by creating reference models and by using the available certification documents. We did not, however, follow the certification process as that would be costly, and to our surprise the certification [30] did not cover the implementation in detail, e.g. omitting the large number of rules related to the ArchiMate language.

3.1 IoT device

The first language we implemented targets the development of applications for an IoT device. Rather than have developers manually write the application code covering the logic, reading sensor data, and communicating with the outside world, we created a DSM solution consisting of two integrated modeling languages and a JSON generator. An application developer can therefore design the application using the concepts of the IoT device, such as its sensors and services, and produce complete code from the models which can then be executed in a device[1].

The core language is based on a state machine that is extended with services and rules of the device. Sensors (movement, pressure, luminance, GPS, temperature etc.) provide various events and data along with some internal housekeeping functions like checking battery level and charging status. The action part of the state machine covers various communications methods with which the device can interact with the outside world (like sending messages to the cloud or SMS to a phone). Fig. 1 shows a small illustrative example developed with the implemented DSM solution: a sauna application that uses temperature and humidity sensors. When the temperature is over 60°C, the application checks the humidity and sends the information to the given phone number. The example is small as only 2 out of over 10 possible sensors are used, and only SMS sending is used. The figure also shows the modeling environment for creating the models, checking them and running generators.

While the first language focuses on implementing a single application, the second IoT language, integrated with the first, allows developers to integrate several applications or modules. This helps to address scalability and manage various models as in Fig. 1.

3.1.1 Parts of the implemented DSM solution. The completely defined metamodel contains 74 elements. As a point of reference, this is about one third of the size of the metamodel of UML 2.5 [20] in the same metamodeling language in MetaEdit+. In addition, 39 constraints were defined in the metamodel. These include how objects can be connected with each other by relationships, how many participants there can be in such connections, and how diagrams can be organized to handle the design of large applications. The most common kind of constraints were regular expressions to ensure that entered property values were correct within the domain, e.g. legal values for humidity, pressure etc., or to make providing a value mandatory. These constraints ensure that the data entered into the model is legal and so can be directly used in code generation. While creating the metamodel and constraints, the metamodel concepts were annotated with descriptions where necessary, so that the modeling tool can provide an integrated help system for the language users.

For the notation, 54 symbol elements were defined. Many of the notational elements were conditional showing different symbol elements depending on the design data entered in the model. In addition, 9 small symbols were defined to be used as icons. These icons are not mandatory — MetaEdit+ will automatically provide icons based on the main symbol — but make the language concepts easier to identify from the editor's toolbar from which the elements are selected. These icons are also used to differentiate the concepts in various views and browsers.

Generators were defined for two purposes: 1) for checking the rules and constraints that were not expressed in the metamodel — mostly incompleteness, like a missing start state or elements that are not connected with the rest of the designs, 2) for producing

[1] https://www.thingsee.com/

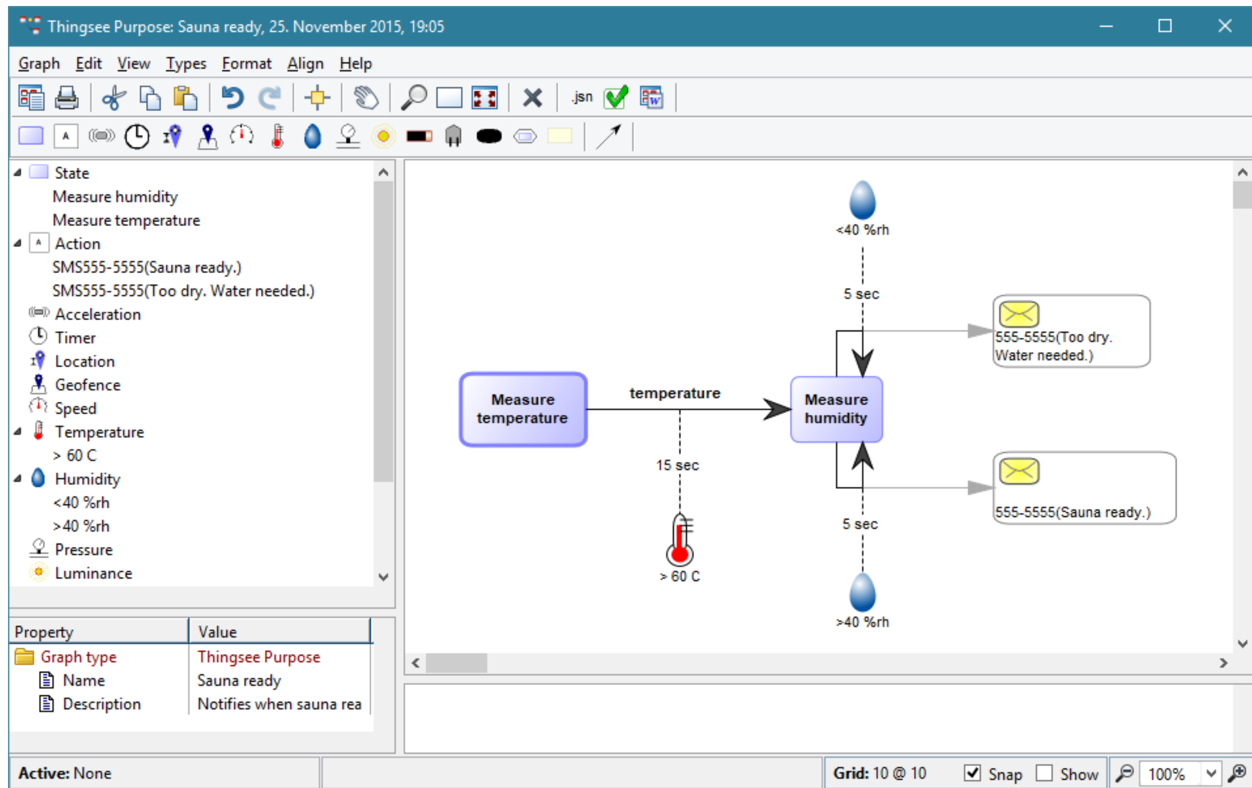

Figure 1: IoT device language in use.

JSON code in a file to be uploaded to the device. For checking, only one generator module (21 LOC in MERL, the MetaEdit+ Reporting Language) was needed because of the work already done in the constraints, whereas the JSON generator consists of 28 generator modules (556 LOC).

Based on these definitions MetaEdit+ provides tool functionality for modeling, managing models, collaborative editing, checking model validity and generating the code. The implementation of the complete language along with its generators is available from https://www.metacase.com/download in the 'IoT' example project in MetaEdit+.

While the language can be considered complete, it could be extended with additional support and guidance mechanisms. These could include examining the specification to inform if the designed use of sensors consumes battery, etc. Nevertheless, we check some of these when relevant for safety, like that the device is not planned to measure temperatures higher than those for which the manufacturer gives a guarantee. This capability is a typical example of providing domain-specific knowledge directly in the DSM solution.

3.1.2 DSM implementation effort. The total implementation effort for creating a ready-to-use IoT modeling support was 3.8 man-days — calculated based on 8-hour work days. The implementation effort spent on different parts of the whole DSM solution is illustrated in Fig. 2. Implementing the metamodel and its constraints and checks took just under one working day. To be precise, based on the automatically collected logs the metamodel was defined

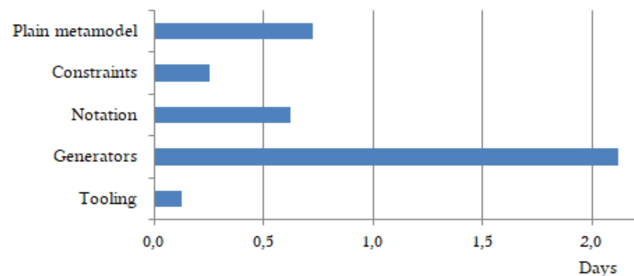

Figure 2: Man-days used to create different parts of the IoT solution

in 5.8 hours. The definition of the 39 rules expressable directly as MetaEdit+ constraints took 1 hour, with an extra 1 hour for those that required generators for checking model completeness and correctness.

The notation covering symbols for the objects, relationships and roles expressed as lines connecting the objects took 5 hours.

Implementing the generator producing JSON took 17 hours. The correctness of the generator was checked by modeling a few sample applications provided by the manufacturer, and comparing the generated code against the provided reference code from the same applications. Later the generator was tested in detail against a large number of small reference applications following a test-driven

approach. The effort to create the test models as detailed in [31] is not included in the generator development effort.

Additional effort to implement the tool support for the language was almost zero, as MetaEdit+ provides the modeling editors, browsers, help system etc. needed based on the definitions. The only tooling related part that was needed was defining the icons used for some of the modeling elements. These were defined in less than an hour making the various language concepts easier to distinguish from each other with a small icon symbol in toolbars and browsers when the default symbol for the notation was not considered adequate.

3.2 ArchiMate

The second language is ArchiMate by The Open Group [28, 29]. The focus of ArchiMate is the representation, communication and analysis of enterprise architecture. The language aims to acknowledge concerns from a variety of stakeholders covering both business and IT. While architects may create the models, everybody should be able to read and comment the models. For this purpose ArchiMate offers a set of entities and their relationships along with suggested icons and other notational elements. Fig. 3 illustrates the use of ArchiMate in the implemented modeling solution when specifying business processes, services and roles related to insurance claims.

ArchiMate models are a combination of traditional data modeling with associations and aggregations, along with process modeling with flows, triggers and access relationships. A particular feature of ArchiMate is that it offers a layered architecture for the models — Business, Application and Technology layers — along with special relationships to relate the elements across the layers. For example, in Fig. 3, the yellow upper part illustrates the Business layer and the four blue elements in the bottom the Application layer. Here two application services, *Customer Data Management* and *Payment Processing* have Serving relationships to elements in the Business layer. The language does not have strict semantics behind the concepts and does not provide real guidance for creating the models: with ArchiMate, the same idea can be modeled in numerous ways.

In addition to the layers, the language provides various additional concepts to describe factors related to motivation, resources and work results. Finally, as the models illustrated in diagrams can quickly become large and there is a need to show different views to different stakeholders, ArchiMate suggests a set of specific viewpoints to the models.

For the implementation of ArchiMate modeling support we thus had a clear specification and there was less need to consider different choices or notational symbols than for the IoT case. Also, as ArchiMate does not support code generation, there was no need to implement generators other than for model checking. Originally, the ArchiMate implementation was created with integration to existing MetaEdit+ metamodels like BPMN for process modeling and ER for data modeling. For the case study purpose these extensions and integrations were excluded and the data collection focused only on the purely ArchiMate parts.

3.2.1 Parts of the implemented DSM solution. The implemented metamodel contains 125 elements. These include 24 abstract elements used to classify modeling elements as in the ArchiMate specification. The abstract elements of the metamodel are not directly visible to language users and are not mandatory for creating modeling support. Also the specification of ArchiMate does not use them when defining the constraints.

Constraints are a notable feature of ArchiMate and in particular the vast number of them ([29], Appendix B[2]): there are over 10,000 rules in the specification and almost all relate to defining which of the 11 relationships types are legal among the 61 objects types.

As the architecture models are not used for generating code, there are no real constraints on how the textual descriptions are entered to the models or even if some data is mandatory. Similarly to the IoT case, descriptions of the language concepts were added to the metamodel so that the modeling tool can provide integrated help while modeling. These covered guidance for all the main modeling elements like its objects and relationships.

For the notation part, 89 symbols were defined, trying to closely resemble the original specification. Most of these symbols are rectangles with a special icon indicating the type of the object and recommended color coding for the layers. Since the specification of ArchiMate includes suggested icons, these too were made, for display in toolbars and browsers. In total 77 icons were defined.

Generators were defined for checking purposes only, addressing some of the constraints that could not be defined in the metamodel. These included how junction concepts can be used to express relationships between more than two elements, grouping modeling objects, and checking that elements connected with composite relationships are legal. These 3 generators were small (42 LOC in the MetaEdit+ MERL language).

Based on these definitions MetaEdit+ provides editors for collaborative architecture modeling, browsers and model management tools, and other expected modeling capabilities. Also in terms of view management, the in-built capabilities of having different views to the same model or single diagram can be applied. The same element can also be shared among different views. In this way data can be displayed appropriately for different stakeholders, or a particular layer can be hidden from the model (e.g. if the focus is on the application layer and others need to be hidden). The implementation of the ArchiMate support is available from https://github.com/mccjpt/ArchiMate.

For reporting and publishing the architecture models as Word or HTML, existing predefined generators of MetaEdit+ could be used. MetaEdit+'s existing generic generators for model checking and calculating metrics could also be used on the ArchiMate models.

3.2.2 DSM implementation effort. Implementation of the ArchiMate modeling solution took almost 4 man-days. This effort was divided into different parts of the language definition as illustrated in Fig. 4. The metamodel was defined in four sessions — in total 12.1 hours. The number of constraints for relationships is huge: 10,760 in total. As adding them manually to the metamodel would be a massive effort, we automated this part of language definition using the generator system of MetaEdit+. The definition of the relationship table ([29], Appendix B) was parsed and binding rules were generated to the metamodel. The few remaining rules that dealt with model structuring into hierarchies were defined manually. The effort to define the generator that parsed the simple textual

[2]http://pubs.opengroup.org/architecture/archimate3-doc/apdxb.html

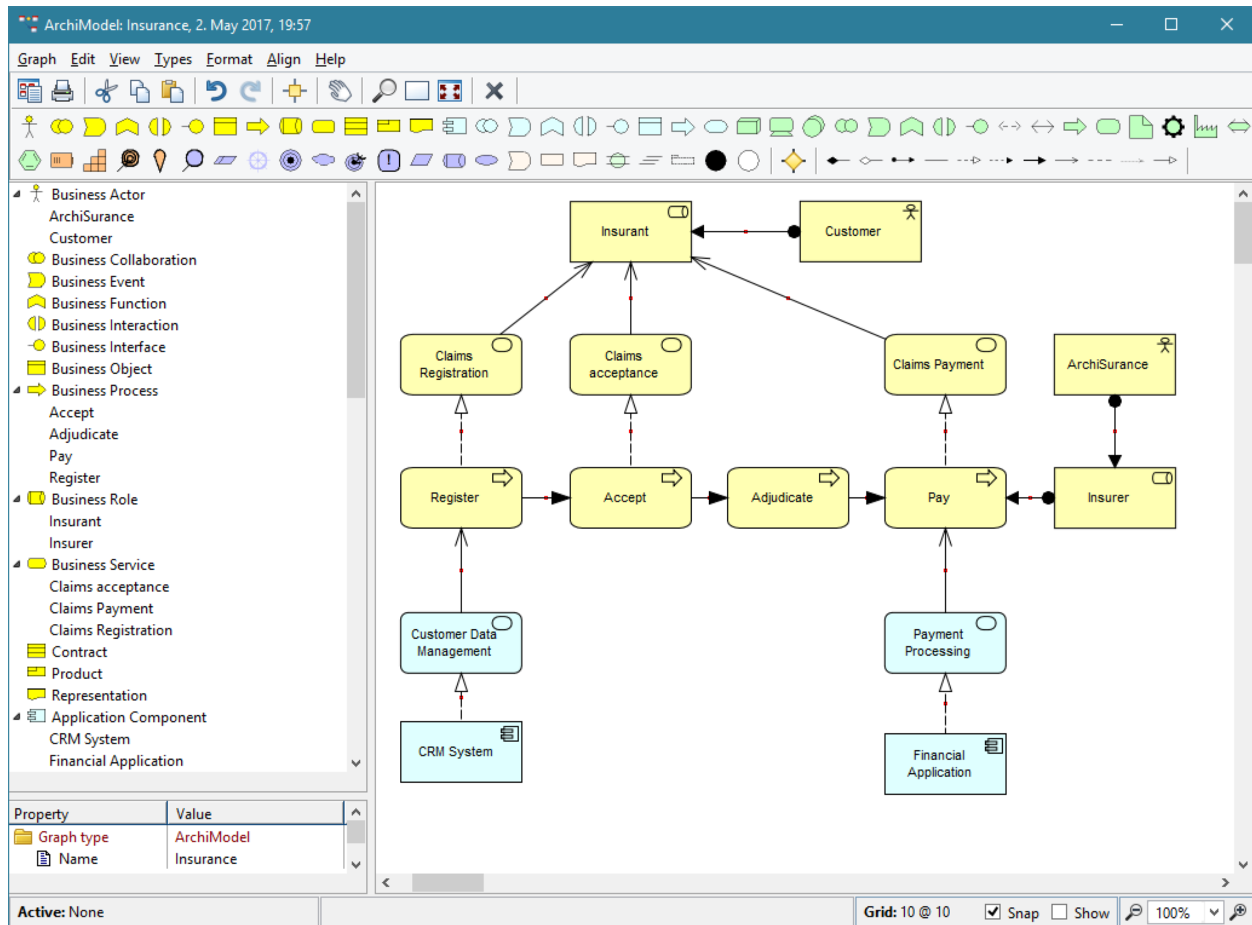

Figure 3: ArchiMate in use.

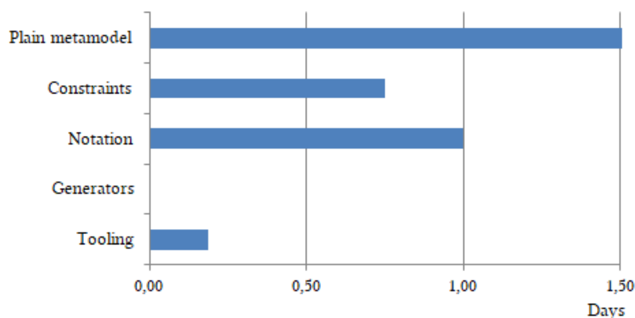

Figure 4: Man-days used to create different parts of the ArchiMate modeling support

rule data and output the constraints in the XML import format for MetaEdit+ metamodels was 6 hours.

While we focus in this study on the initial creation phase, the automated creation of language constraints can be relevant during the maintenance. For example, the ArchiMate language is created by the Open Group in partial definitions without using test or reference data to ensure its correctness. Therefore its earlier versions have had many errors. For example, the latest major release, ArchiMate 3.0 [28], had hundreds of errors in its relationship definition, and version 3.0.1 [29] provided corrections to these. With the automated constraint definition capability, all the constraints of the newer version could be updated to the metamodel by simply running the generator.

The notation, covering symbols for the objects, relationships and roles as well as the icons, was defined in 8 hours. To avoid creating a large number of very similar notational elements, we used the symbol library and template mechanism provided by MetaEdit+. Fig. 5 illustrates this in the definition of the notation for the Technology Interface element. The template in the top right corner — shown with dotted lines and selected in the MetaEdit+ Symbol Editor — retrieves the correct icon from the symbol library and positions it correctly while keeping it from scaling with the rest of the symbol. The same icon is then used for other interface elements of ArchiMate, namely Application Interface, Technology Interface and an abstract External Active Structure Element. This pattern for defining the notation was repeated for other ArchiMate elements like its various functions, services and collaborations. In total 29

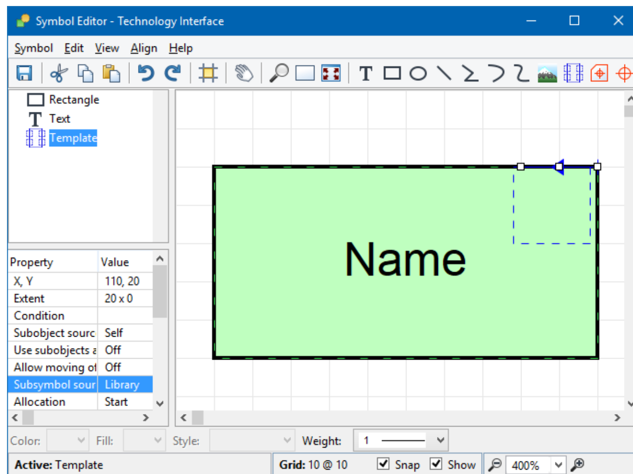

Figure 5: Defining notation for Technology Interface.

notational symbols were reused. This not only made creation of the notation easier but helped to ensure that icons are used in a consistent manner. Since there were no generators, and icons created for modeling elements could be used directly as icons for the toolbar, there was very little additional development effort needed: 1.5 hours in total.

4 DEVELOPMENT EFFORT IN REPORTED INDUSTRY CASES

While our two case studies focused on languages of a realistic size, they were developed in a controlled environment without a typical industry setting, which would normally include reporting, resource allocation and other organizational factors. To compare our experience with others and to examine the development effort in other industry cases we inspected public data — a paper or a presentation — given by other language engineers. We included the cases where the effort was clearly stated, in the paper or in the related presentation, and the language focused on a specific task. So cases where companies had developed multiple generators have been left out — unless they have separated the data on the effort to make comparison possible.

Fig. 6 summarizes the implementation efforts for cases detailed below. In all cases the actual language development — and use — was performed with the same tool, MetaEdit+. This excludes the impact of tooling from the comparison. In 4 of the 8 cases, the language development was performed by the companies’ own language engineers, and they also measured the DSM creation time. In 3 cases (Heating remote control, Military radio systems testing, Blood separators), the main language engineering effort was by an external consultant. In 1 case (Voice control), the main language engineering effort was by a consultant from MetaCase (but not one of the authors). As with our case studies, in all the industry cases the total language engineering team was small: 1 or 2 people.

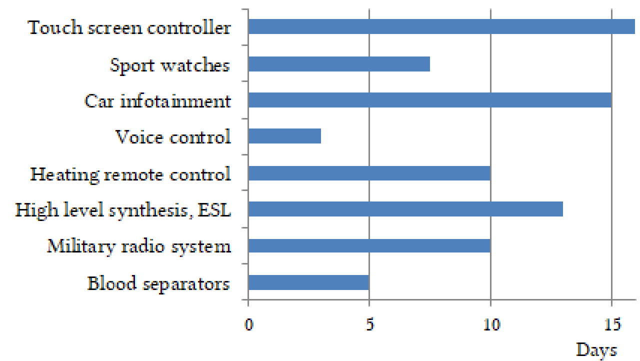

Figure 6: Man-days used to create DSM solutions in different cases

4.1 Review of reported industry cases

4.1.1 Touch screen controller at Panasonic [25]. The language focuses on developing applications for home automation in an embedded device. A generator reads the models created and produces C and HTML for a Linux based touch screen. Panasonic reported their total development effort as 16 man-days, without differentiating the effort on the various elements of their DSM solution. However, later development of a second generator targeting a microcontroller using the same models was reported to take 3 man-days. This generator produces C code with build/flash/run workflow automation.

4.1.2 Sport watches at Polar [14]. The DSM solution focuses on embedded applications in a sports watch. The models specify the application UI, navigation and localization, generating C code which is typically about half of the total code in a final product. Polar reports having used 60 man-hours for developing their DSM solution.

4.1.3 Car infotainment system by OEM [[2], talk at Leipzig]. A car manufacturer developed a DSM solution for specifying end-user applications for infotainment systems. The models cover static UI elements, UI logic and navigation and calls using the services (radio etc.). The generator produces Java for simulation and concept design. The OEM invested 3 man-weeks in total for creating the DSM solution: 1 week for the language and 2 weeks for the generator.

4.1.4 Voice control systems for home automation [11]. The supplier focused on machine-to-machine communication for the energy and home automation sector. The DSM solution focused on specifying voice commands and related control logic used via telephone lines. The DSM solution was implemented in 3 man-days during a two-day workshop with a MetaCase consultant and an expert of the domain for the domain analysis. The first day of the workshop produced a language that was too generic to permit code generation, resulting in a need to start from scratch under time pressure. The second attempt took 1.7 hours for the modeling language, and the generator producing assembler code for an 8-bit microcontroller took just over 2 hours.

4.1.5 Remote control for heating systems at Ouman [23]. The DSM solution specifies a control application for mobile phones, with a generator producing Python. An external consultant developed the solution in collaboration with one person at Ouman. The consultant made the implementation alone and a person at Ouman tested it to provide feedback. The development effort by the consultant was 2 man-weeks.

4.1.6 High-level synthesis for virtual platforms at Profound [22]. The DSM solution specifies the block structure for image processing systems and generates SystemC for high-level synthesis. For creating the first version, Profound used 4 man-days for the metamodel, 3 days for the notation and 5 days for the SystemC generator. The implementation was by one person who also mastered the domain of the synthesis.

4.1.7 Testing military radio system at Elektrobit [24]. For testing VoIP networks and related terminals, a DSM solution specifies test cases and larger-scale test logic. The generated target code is the TTCN3 test scripting language. The DSM development effort was 2 man-weeks. One additional week was used later to implement feedback functionality from the test environment back to the modeling tool, annotating models to visualize the execution of tests. The language testing was done later by another person creating an existing reference solution as a test case.

4.1.8 Blood separators by a supplier [5]. A medical device manufacturer has a product line of blood separators. As a part of development solution, including hardware and software, a consultant from a MetaCase partner developed a modeling language for the product line. Generators were developed to produce IEC61131 program code and configuration code. The consultant reports an investment of one man-week to create the DSM solution.

4.2 Comparing and combining the cases and industry reports

One main difference between the two cases and industry experiences is that in the latter the domain knowledge has already been available — at least the language engineers have had past experience of the problem and implementation domain, whereas in both case studies the domains were unfamiliar. Conversely, the cases were performed by an experienced language engineer, familiar both with language engineering in general and MetaEdit+ in particular. Similarly, the cases started from an already existing set of language concepts, whereas the industry experiences first had to identify the domain concepts.

The industry cases show an average of just under 2 weeks' development effort to create a DSM solution — 9.8 days, ranging from 3 days to 16 days with a standard deviation of 4.6 days. Our two case studies with their 4 man-day development efforts fit into this range from the industrial cases — albeit at the lower end. The industry cases using purely companies' in-house resources took on average 12.6 days, whereas those using an external consultant took on average 7.0 days. The 3 cases performed by MetaCase language engineers (1 from industry and 2 reported here) averaged 3.4 days.

Although by no means certain, these figures would seem to support a hypothesis that the most experienced language engineers may be a few times faster than a first-timer: a significant 'transfer of learning' effect, indicating that there are domain-independent elements that can be learned once and applied more efficiently the next time around. The question remains open as to how much of this is generalised learning about language creation, and how much is learning MetaEdit+. In any case, a first-timer both at creating a language and using MetaEdit+ is able to create a language on average in 2–3 man-weeks.

Unfortunately most of the industry cases found do not break the development effort down into the different parts of the DSM solution. The available data indicates that when implementing a DSM solution with a generator for one output language, building the generator took about 40–60% of the total effort (42% in High level synthesis, 54% in the succesful second attempt at Voice control systems, 55% in IoT). Excluding generators, the notation took about 30–45% of the total time to build the language (29% in ArchiMate, 36% in IoT, 43% in High level synthesis).

Looking at the various phases and artefacts, perhaps the largest variation between cases is on the earliest stages: domain analysis, deciding the boundary and level of the language, and coming up with the main idea of the language. No separate times were gathered for these tasks, and indeed they are often spread across the metamodeling and notation phases. As seen in the Voice control systems case, even a relatively experienced language engineer can choose a wrong path and be forced to restart from scratch. Having a tool that allows quick, incremental building of a metamodel during these early stages can help. Making things concrete reduces the time lost, allowing early identification of problems and experimentation with various approaches.

The size of the resulting language had a surprisingly small effect on the effort required. We have data on language size from three of the industrial cases (sadly mostly not public). Although more data would be needed, it seems that effort increased less than linearly with size. Even with the largest language, nearly twice as large as UML, there is certainly no evidence of slowing caused by hitting problems of scalability or complexity. On the larger languages it seems that patterns develop for extending the language further: a new concept is often quick to add, being one more member of a set of similar existing concepts.

5 OTHER RESEARCH COMPARING EFFORT

We are not aware of many systematic studies comparing the development effort across several DSM solutions. Mostly, studies reporting development effort focus on a single case, and differences in domain, language, generation need and tools make it hard to compare across individual cases.

Some cases with comparisons do not collect effort directly, but via a proxy such as lines of code, sometimes even for different code languages. For example, in [18], a language engineer implemented the same modeling language twice — with a difference in the language definition mechanism applied. From the reported language specification we can detect that defining a UML profile with OCL constraints requires twice as many lines of constraint specification than implementing the modeling language with a metamodel using constraints offered by MetaEdit+.

Similarly in [7], several tools were given the same set of tasks, and the length of their specification in lines of code was compared.

Unfortunately the lines of code were in different languages for each tool, and the tools could choose which parts of the tasks to include, weakening the accuracy and readability of the comparison. The importance of open publishing of data was however highlighted, when further analysis in [12] was able to recover some of the information by graphing tasks completed against lines of code.

Somewhat better than comparing lines of code from different, generally unfamiliar languages is when lines of code in a known language are used as a proxy for time. This was the approach taken in [10], which took a known Eclipse GEF case in Java, and used COCOMO to estimate based on its 10,000 lines of code that its development effort was around 13 man-months (2,000 hours). This was then compared against implementing the same language in MetaEdit+, which took 1 hour. This was one of the few cases that broke down the development time into its parts: 15 minutes for the abstract metamodel and 45 minutes for the notation.

Time is a better measure of effort than lines of code. De Smedt [26] built the same simple language and its transformation and simulation support with his department's AToM3 in 13 hours and with MetaEdit+ in 11 hours. The experiment showed the value of picking appropriate tools for the task: AToM3 has a focus on simulation, making that part easier than with MetaEdit+ where the API had to be used. Similarly, an attempt to use Poseidon was forced to stop early on because of the lack of support for transformation.

In [21], two different DSM solutions are created for the warehouse automation domain with a focus on evaluating decisions made during language design. In addition to creating modeling languages, two generators were defined to produce complete automation software, including the source code in IEC 61131-3 and the visualization of the automation system. These DSM solutions were both defined in MetaEdit+ but following different language design patterns and decisions. The resulting DSM solutions were then applied to develop the same warehouse system functionality. The implementations were made by students who did not have prior experience of DSM or the domain of automation systems. The effort to develop complete DSM solutions as student work was on average 33 man-days.

Finally, research indicates that tools can make a remarkable difference to the effort needed to create DSM solutions. An empirical study [13] applied a controlled laboratory study in which a portion of the same language, BPMN, was implemented in 5 different tools. Thus this study resembles the case of implementing ArchiMate based on an existing language specification. This study shows large differences in the development effort depending on the tool used, ranging from 0.5 days to 25 days. The smallest development effort was with MetaEdit+ 4.5 [16] and the largest with the Eclipse Graphical Modeling Framework 2.4.0 [8].

5.1 Integrating other research with our results

Perhaps most remarkably in [13], the tool with the second fastest development effort was already an order of magnitude slower than with MetaEdit+. One possible factor would be if the experiment subject using MetaEdit+ was simply a much better language engineer than those using the other tools: our cases in this paper show a factor of 3.75 between the most experienced language engineers and first-timers. To test this, one of the current authors implemented the same BPMN subset with the same MetaEdit+ version. That reimplementation took 45 minutes, over 5 times faster than the original, placing the original experiment subject at a similar skill level to other first-timers.

When the same language was made with different tools [13], by language developers of a similar skill level, effort thus varied by a factor of 50. When the same tool was used to make different languages here, effort varied by a factor of 5. A factor of 5 seems relatively modest, given the spectrum of cases covered here — the size and complexity of domain and language, need for generation, and experience of language implementer (with the tool, language creation in general, the problem domain and the solution domain).

With a factor of 50 based on the tool, and a factor of 5 on all other factors combined, the tool choice seems likely to cause the largest differences in the development effort. This effort translates into the cost of language engineer time, which in an industrial setting far exceeds factors of tool price. This indicates that teams creating domain-specific languages should pay special attention to the tool selection and be aware of that when budgeting for resources and time.

We look forward to other case studies and language creation efforts in the industry reporting data on the development effort, time used, size of the team, etc. Establishing good metrics for collecting and analyzing the effort is challenging, particularly for industrial cases. Ideally research would collect data for different elements or phases of language creation (abstract syntax, constraints, notation, generators and tooling).

6 CONCLUSIONS

We compared the effort to create complete DSM solutions across ten cases, combining evidence from two complementary sources: detailed case studies and reported industrial experiences. Case studies provided us detailed data on how much effort is spent on creating different parts of the DSM solution. Although the two case study languages are different, the collected data has similarities: Creating the abstract syntax in a metamodel seems to take more time than creating the concrete syntax used for the notation.

The majority of public industry cases with this tool (or indeed others) did not indicate the effort at all. Those found, however, show that the effort to create a DSM solution with MetaEdit+ is modest: ranging from 3 to 15 man-days, with an average of 10. This is in line with our two case studies with a 4 man-day development effort — around the lower end of the industry cases. It is also worth noticing that the size of the effort was quite similar among the DSM solutions, even though they addressed different domains and were created by language engineers with different backgrounds and experience levels. As with our case studies, the actual implementation was done in all industry cases by one or two people. This indicates that large teams are not needed for language development (at least with this tool), but naturally a larger number of language users is welcome during the testing and incremental development of the language and DSM solution.

Unfortunately most of the industry cases found did not break the development effort down into different parts of a DSM solution. The available data from industry cases indicates that when

implementing a DSM solution with a generator, building the generator takes about twice the effort to creating the modeling language (metamodel, constraints and notation). Obviously 10 cases is not enough to generalize the development effort, in particular when having differences on the domains addressed and language engineers' experience, yet they indicate that the investment need not be large.

Once a DSM solution is available, language users have been able to start creating models and generating the code rather quickly. The high productivity of DSM provides a good return on investment, and the effort used to develop the language is thus paid back quickly — as data for ROI calculations indicate [14, 21, 24, 25].

While we focused here on the language creation phase, we acknowledge that as in all software development, the maintenance phase is normally greater in terms of total effort. However, the effort to update the language, generators and tooling should generally be proportional to that of creating the language in the first phase. A major difference, however, is that during maintenance any changes to the language will also require that existing models be migrated to newer language versions. Tools vary widely in the work required for this, and future empirical research is needed to quantify the effect of that difference over the whole lifecycle.

We welcome future research applying empirical research methods to language engineering and language creation — as well as measurements from industrial cases creating languages, generators and tools. Researchers can repeat and validate this study following the same taxonomy of parts of the DSM solution and report their efforts. The scope of the research can also be extended by addressing other domains and other kinds of modeling languages. Since we focused on one tool and research indicates that different tools have a large influence on the implementation effort, we welcome further research addressing a wide variety of tools, or identifying which parts of DSM solutions are especially faster or slower with certain tools.

REFERENCES

[1] Ronan Barrett. 2015. 5 Years of 'Papyrusing', Ericsson Modeling Days, Kista, Sweden, November 9-10, 2015. Retrieved from https://docs.google.com/presentation/d/1nR6GRBsS2Ad3OQf8ldMO8MydFEeWFsFd8WqjQ23Bz8s

[2] Carsten Bock. 2006. Visuelle domÃďnenspezifische Sprachen — Der Schlüssel zur modellgetriebenen Entwicklung von Mensch-Maschine-Schnittstellen?, Model-Driven Development and Product Lines: Synergies and Experience, Leipzig, Germany, October 19-20, 2006.

[3] Francis Bordeleau. 2014. Papyrus and Open Source Modeling — Status, Strategy, and Plan. Presentation at Ericsson Modeling Days, Kista, Sweden, 4 November, 2014

[4] N. Brouwers, R. Hendriksen, K. Kahraman, J. Kouwer, and J.-P. Tolvanen. 2016. Industrial use of domain-specific modeling, DSM workshop, SPLASH. Retrieved from http://dsmforum.org/events/DSM16/

[5] Verislav Djukić, Aleksandar Popović, and Juha-Pekka Tolvanen. 2014. Using domain-specific modeling languages for medical device development, Embedded.com, March 08, 2014.

[6] DSMForum.org. 2018. http://dsmforum.org/cases.html

[7] S. Erdweg, T. van der Storm, M. VÃűlter, M. Boersma, R. Bosman, W.R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P.J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. 2013. The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge. In Proceedings of the Software Language Engineering conference, Springer, 2013.

[8] Richard Gronbach. 2009. Eclipse Modeling Project. A Domain-Specific Language (DSL) Toolkit. Addison Wesley.

[9] Steven Kelly, Kalle Lyytinen, and Matti Rossi. 1996. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In Proceedings of CAiSE'96. Springer.

[10] Steven Kelly. 2004. Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In Proceedings of the OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development. http://www.softmetaware.com/oopsla2004/mdsd-workshop.html

[11] Steven Kelly and Juha-Pekka Tolvanen. 2008. Domain-Specific Modeling: Enabling Full Code Generation. Wiley.

[12] Steven Kelly. 2013. Empirical Comparison of Language Workbenches. In Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling. http://dsmforum.org/events/DSM13/

[13] A. El Kouhen, C. Dumoulin, S. Gérard and P. Boulet. 2012. Evaluation of Modelling Tools Adaptation. CNRS HAL hal-00706701. http://tinyurl.com/gerard12

[14] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. 2009. Evaluating the use of domain-specific modeling in practice. In Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling. http://www.dsmforum.org/events/DSM09/Papers/Karna.pdf

[15] M. Mernik, J. Heering, and A. Sloane. 2005. When and How to Develop Domain-Specific Languages, ACM Computing Surveys, 37, 4.

[16] MetaCase. 2006. MetaEdit+ 4.5 User's Guide. http://www.metacase.com/support/45/manuals/

[17] MetaCase. 2017. MetaEdit+ 5.5 User's Guide. http://www.metacase.com/support/55/manuals/

[18] K. Mewes. 2009. Domain-specific Modelling of Railway Control Systems with Integrated Verication and Validation. Ph.D. thesis. University of Bremen.

[19] P. Mohagheghi, W. Gilani, A. Stefanescu, M. Fernandez, B. Nordmoen, M. Fritzsche. 2011. Where does model-driven engineering help? Experiences from three industrial cases. Software and Systems Modeling 12, 3, 619-639.

[20] Object Management Group. 2017. Unified Modeling Language Version 2.5.1.

[21] C. Preschern, N. Kajtazovic, and C. Kreiner. 2014. Evaluation of Domain Modeling Decisions for Two Identical Domain Specific Languages, Lecture Notes on Software Engineering 2, 1 (Feb. 2014). DOI: https://doi.org/10.7763/LNSE.2014.V2.91

[22] Profound. 2018. Construction of cooperative design environment of software / hardware using single-threaded ESL tool. Retrieved April, 2018 from http://www.profound-dt.co.jp

[23] Olli-Pekka Puolitaival. 2011. Home Automation DSL Case, Presentation at Code Generation Conference, 2011.

[24] Olli-Pekka Puolitaival, Teemu Kanstrén, Veli-Matti Rytky, and Asmo Saarela. 2011. Utilizing domain-specific modelling for software testing, 3rd Int. Conference on Advances in System Testing and Validation Lifecycle.

[25] Laurent Safa. 2007. The Making of User-Interface Designer, a Proprietary DSM Tool. 7th OOPSLA Workshop on Domain-Specific Modeling. http://www.dsmforum.org/events/DSM07/

[26] P. De Smedt. 2011. Comparing Three Graphical DSL Editors: AToM3, MetaEdit+ and Poseidon for DSLs. University of Antwerp. http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/201011/projects/Philip.DeSmedt/report/report_PhilipDeSmedt.pdf

[27] J. Sprinkle, M. Mernik, J.-P. Tolvanen, and D. Spinellis. 2009. What kinds of nails need a domain-specific hammer?. IEEE Software, (July-Aug, 2009).

[28] The Open Group. 2016. ArchiMate 3.0 Specification.

[29] The Open Group. 2017. ArchiMate 3.0.1 Specification.

[30] The Open Group. 2017. Architecture Tool Certification: ArchiMate 3 Conformance Requirements.

[31] Juha-Pekka Tolvanen. 2016. Applying Test-Driven Development for Creating and Refining Domain-Specific Modeling Languages and Generators, In Proceedings of 16th Workshop on Domain-Specific Modeling, Amsterdam.

[32] J. Whittle, J. Hutchinson, and M. Rouncefield. 2014. The State of Practice in Model-Driven Engineering, IEEE Software 31, 3 (May-June, 2014)

[33] Markus Voelter. 2013. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace.