Sketch-based Metamodel Construction
A Literature Review

Lucas Heer
University of Antwerp
lucas.heer@student.uantwerpen.be

February 13, 2018

Abstract

Domain-Specific Modeling Languages (DSMLs) are increasingly used by system engineers to specify and document both the structure and behavior of complex software systems. Compared to general purpose programming languages, they present numerous advantages to the engineer, such as improved expressiveness, intuitive syntax tailored to a specific domain, and increased level of abstraction. However, the development of a DSML is often carried out in a top-down manner with the definition of the metamodel for the language as first step. In many cases, this is a complicated and error-prone process and does not reflect the reality of software engineers that prefer to initially draw informal instance model sketches either on paper or in common drawing tools such as Microsoft Visio or Dia. These sketches are merely considered as first drafts and serve no further purpose than expressing ideas in the early stages of development. This traditional approach also introduces an expertise gap since domain experts rarely are experienced enough to create high-quality DSMLs without the help of a dedicated language engineer. To overcome the shortcomings of the top-down DSML design process, several approaches have been proposed to derive a language definition solely from informal sketches that are typically found in the very beginning of the modeling process. This promises to ease the creation of DSMLs, ultimately making model-driven engineering of complex software systems more accessible for non-experts. This paper discusses the current state of research and analyzes proposed solutions for designing domain-specific languages in a bottom-up way. Individual challenges of this approach are identified and the solutions are examined to discover shortcomings and areas where further research is needed.

1 Introduction

One of the major challenges in modern software development is to master the growing complexity of software-intensive systems. With the increasing amount of requirements that are imposed on such systems, engineers try to find new ways to make the development process more efficient and less error-prone. A lot of effort has been spent to reduce the so-called accidental complexity. As opposed to the essential complexity, which is inherent to a system and cannot be reduced, the accidental complexity is attributed to inefficient development techniques that are considered a hindrance to the developer [10]. This chapter describes fundamentals that act as basis for the subsequent chapters and presents the tools and approaches that were investigated in the scope of this review.

1.1 Background

The most predominant approach to reduce the accidental complexity of software engineering is to raise the level of abstraction at which the structure and behavior of a system is described. Historically, this technique has led to the establishment of high-level programming languages such as Java or C++ that try to hide irrelevant and unimportant aspects that are not needed to solve a problem. With Model-Driven Engineering (MDE), another trend has emerged to counter the growth in complexity of software-intensive systems. MDE allows to raise the level of abstraction above general-purpose programming languages. Instead of code, models are used to describe a system's structure and behavior. The advantages are multifold: the accidental complexity reduced by increasing expressiveness, formal verification is easier on higher levels of abstractions and various studies suggest a large productivity improvement by using MDE techniques [49]. Beside general-purpose modeling languages such as Petri Nets [73] or Statecharts [40] that can be used for a broad range of domains, the class of Domain-Specific Modeling Languages (DSMLs) allows for MDE to be tailored to the particular domain of the system. Because of that, the structure of a DSMLs has to be defined first by a metamodel. The metamodel specifies all available concepts in the language and imposes restrictions on how these concepts can be used [51]. After that, models can be instantiated from the metamodel. Therefore, models that conform to the metamodel are also called instance models. The creation of a DSML is often a complex and time-consuming task that requires in-depth knowledge about metamodeling and specialized tools that have a steep learning curve [71]. Typically, it is carried out by a language engineer that uses input from a domain expert to design an appropriate formalism. This includes the abstract syntax (the structure of the concepts), the concrete syntax (how the language concepts are represented) and the semantics (the actual meaning of the concepts) [41]. For the concrete syntax, either textual or graphical notations are established. The abstract syntax is typically a data structure that describes the semantically relevant language concepts and their relations. This is usually done by the metamodel of the language. Finally, the semantics of a language can either be defined by mapping the behavior to another known formalism such as Statecharts or Petri Nets [63], in which case it is called denotational semantics or by explicitly giving the execution rules. In this case, it is called operational semantics [62].

Within the MDE community, the most commonly used process to develop a DSML from scratch is executed in a top-down manner [61][85]. It typically starts with gathering and analyzing requirements from the domain experts. The domain experts are engineers that want to use a DSML to describe a system, but do not necessarily have the expertise to create such a language. Therefore, dedicated language engineers define the abstract and concrete syntax and the semantics based on the input of the domain experts. After the development and testing of the language is finished, the domain experts verify if it meets their initial requirements. If not, another iteration of the development cycle can be executed. Figure 1.1 depicts the different roles and activities that are involved in the top-down DSML creation process.

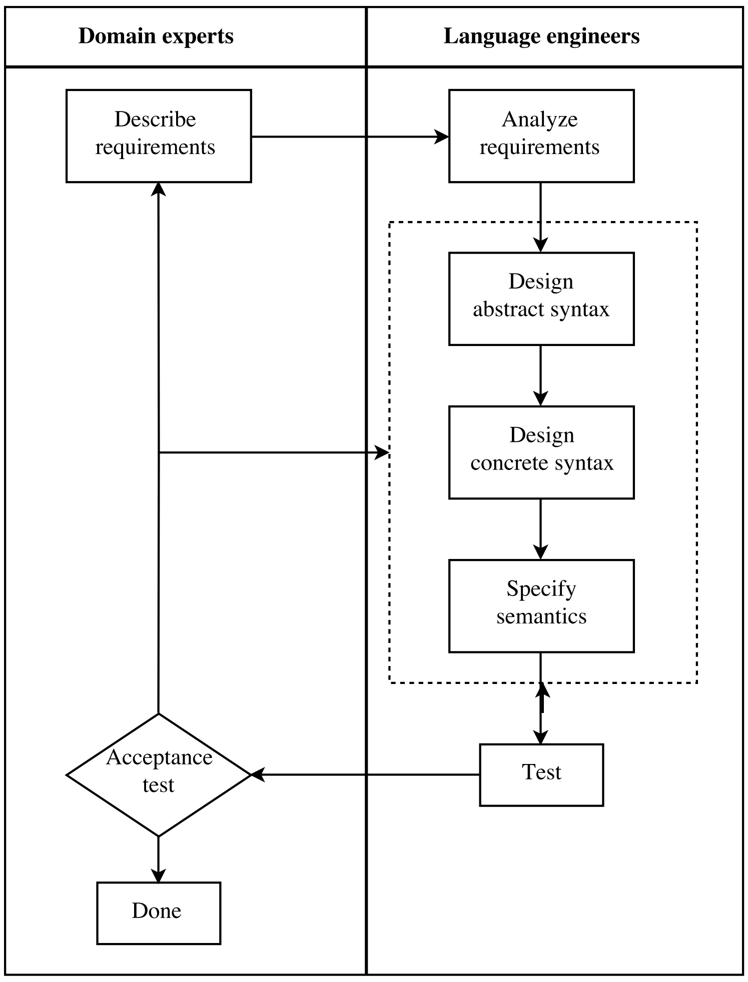
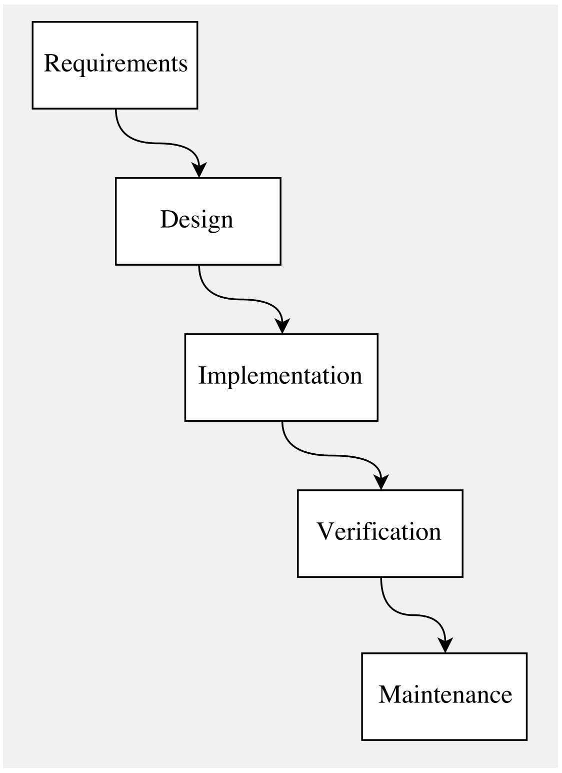


Figure 1.1: Top-down approach for DSML design, adapted from Cho et al. [15]

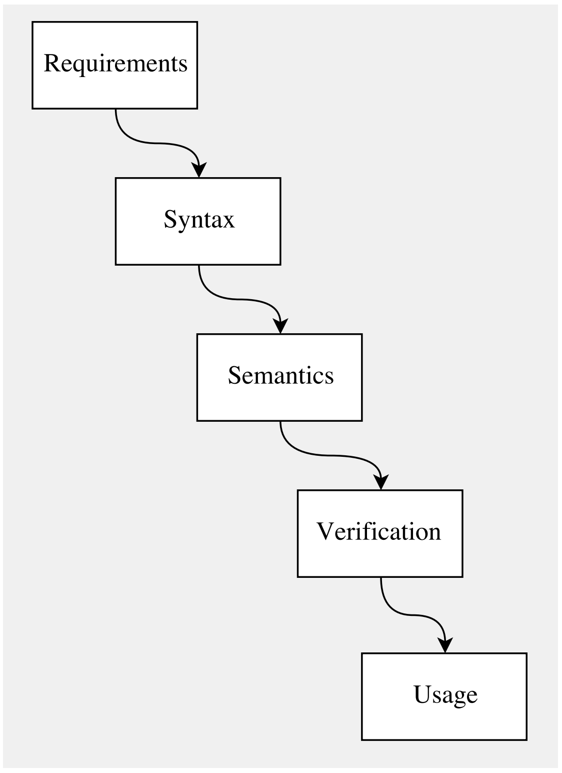
The described DSML design process exhibits major flaws that can be attributed to the centralization of the metamodel construction and to its strict top-down nature. Most noticably, the metamodel that describes the abstract syntax is directly derived from the requirements of the domain experts. However, this approach is counter-intuitive for the domain experts since they may not be aware of all requirements yet and might therefore fail to state them in a concise and complete manner. In reality, they often start drafting example models first and then abstract them into elements and relations of the envisaged DSML to clarify the requirements [57]. Research has also shown that such informal sketching is a vital aspect in creative problem solving methods which are often used in early phases of the system development process [33][91][13]. Beaudouin-Lafon et al. describe pen and paper prototyping as the fastest, cheapest and most widespread prototyping method for interactive system design [8].

Another problem stems from the strict separation of domain experts and language engineers. The creation of a DSML inherently requires domain knowledge, but domain experts often do not have the required competence to design a metamodel for the language [46]. This results in an expertise gap between the domain experts and the language engineers that introduces communication problems and might be difficult to overcome [18]. It also shifts a lot of the work to the language engineer and creates an imbalance in the workload of the involved engineers [56]. There is currently very little tool support for creating a DSML without being familiar and experienced with the concepts of metamodeling. This is especially important since MDE tools are still considered a major hindrance for wide-spread industrial adoption [90][71].

The described approach to design DSMLs bears strong similarities with early approaches for software development processes and also shares their shortcomings. The iterative, but somewhat rigid process depicted in figure 1.1 is comparable to the waterfall model, first formally described by Royce [82]. Characteristic to this development process is that each stage must be executed in its entirely before moving on to the next. Typical stages include requirements analysis, design, implementation and testing.



(a) Waterfall model for systems engineering



(b) Waterfall model adapted for DSML design

Figure 1.2: Comparison between traditional waterfall model and adapted waterfall model for top-down DSML creation

Figure 1.2 shows how the software development waterfall model can be adapted for the construction of DSMLs. Similar to the process shown in figure 1.1, it assumes a linear succession of the individual steps, starting from the requirements. Therefore, the shortcomings of the waterfall model can also be applied to the currently established top-down process of DSML creation and can be summarized as:

1. **Requirements first:** Both the waterfall model and the top-down process depicted in figure 1.1 assume that all requirements for the software artifact to develop are completely known beforehand. However, it is nowadays a generally acknowledged fact that often, requirements are initially incomplete and may be contradictory and inconsistent [77].

2. **Linear and rigid:** The waterfall model imposes a strict sequence of the individual activities and does in its original form not allow for any kind of iterations or incremental development. It is therefore difficult and expensive to react to changing requirements in later phases of the development. Also, if different developer roles for the activities are assumed, the sequential nature of the waterfall model wastes a lot of time, for example by letting the test engineers idle during the initial requirement engineering phase.

3. **Late delivery and feedback:** The waterfall model only produces results in the late phases of the process. Therefore, users and stakeholders can give feedback only once the product is finished. This may lead to expensive and long re-iterations of the complete activities.

Altogether, these issues have resulted in the establishment of methodologies that belong to the class of iterative and incremental development methods [53]. Most prominently, the family of agile development methods promises to remedy the shortcomings of traditional approaches by letting the requirements and product iteratively evolve over time [65]. One of the key elements is the ability to quickly react to changing user requirements by shortening the time it takes to execute one iteration of the development process. Another element is intensive communication within the team by regular meetings and regular feedback from the stakeholders of the product to develop [75]. Over the recent years, it has been discussed how the agile methodology can be combined with MDE. This combination is commonly referred to as ”Agile Model-Driven Engineering“ (AMDD) and has first been described by Ambler [2][1]. Since then, several concrete processes have been proposed [66] as well as a multitude of applications [27], with preliminary research and industrial case studies showing promising results [1][96]. However, these approaches mainly focus on the agile development of software-intensive systems with general-purpose modeling languages and not the DSML construction process itself.

The most promising trend to overcome the issues of the linear top-down DSML creation process is the field of example-based metamodel construction. By providing a set of example instance models, it is tried to automatically infer a metamodel to which all these instance models conform to. This approach bears similarities with the Programming by Demonstration movement (PbD, sometimes also called Programming By Example, PbE), a technique that appeared in software development research in the mid 1980s [37][55]. Here, the main objective is to make computer programming more accessible to end-users: If the user knows how to perform a task on a computer, that should be sufficient to create a program that performs this task. Hence, it should be unnecessary to learn an actual programming language. By recording what the user does, the computer "comes up" with a program that corresponds to the user's actions. The main challenge hereby is how to infer the user's intent from his actions. Since the context and data may change between executions of the program, it is important to acquire a correct set of generalizations for the user's actions. Often, the intent can be derived by inspecting and comparing multiple user recordings or letting the user manually select the right intention from a list of possible candidates [23]. Several uses for PbD have been proposed, most noticeably in the field of robotics [9] and machine learning [54].

1.2 Bottom-up metamodel construction

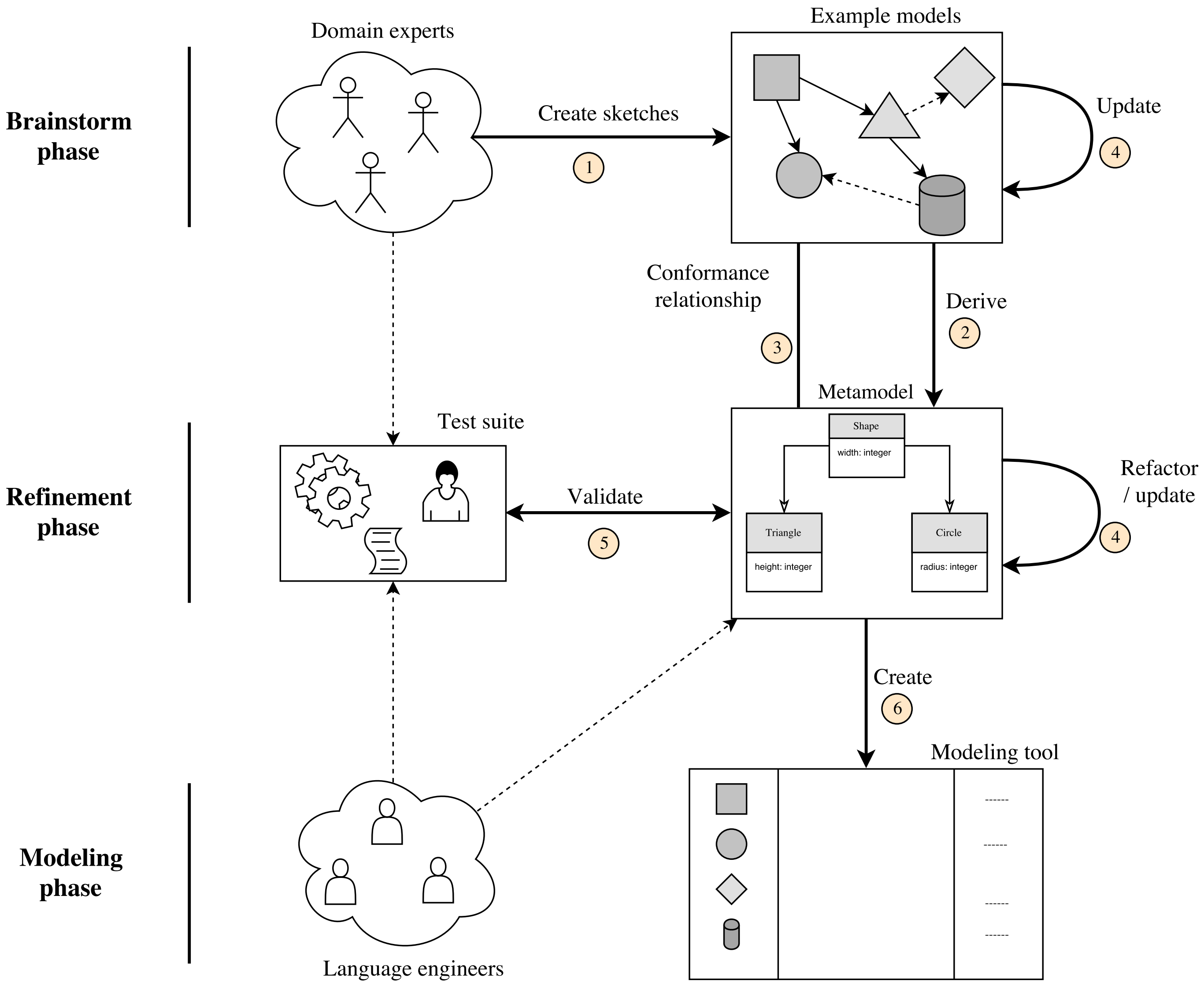


Figure 1.3: Bottom-up, example-driven approach for DSML design

The principal idea of PbD has been transferred to the field of DSML construction. Figure 1.3 gives a generic overview of such a bottom-up process, the different phases it incorporates and the roles of the domain experts and language engineers. The whole process is divided into the brainstorming, refinement and modeling phase. As depicted, the initial brainstorming phase consists of the domain experts thinking about the requirements of the DSML and expressing their ideas as a set of fragmentary example models. These example models do not necessarily have to be complete, but can also represent a single aspect of the DSML only. Most commonly, these example models are of graphical nature and are drawn by hand on some kind of electronic device. As soon as one or more example models exists, a metamodel is inferred (step 2). This metamodel can then be altered, either by hand, by automatic optimization like refactorings or by changing the set of example models and deriving a new metamodel from them (step 4). Throughout the whole refinement phase, the conformance relationship between the metamodel and the example models must be maintained, preferably in an automatic way (step 3). Note that the refinement phase of the metamodel can be carried out by both the domain experts and the language engineers, with the latter usually working on the metamodel level instead of the example models. Step 6 is used to verify if the metamodel meets certain defined quality and requirements. If it does, the metamodel can be either imported into or used to create a modeling platform. If it does not pass the tests, another cycle of the refinement phase can be executed, either by directly manipulating the metamodel or indirectly by changing the set of example models to reflect the necessary changes.

This bottom-up process also lessens the burden of the language engineers which no longer plays a central role during the DSML development process. Rather than actively carrying out the requirements analysis and the subsequent language elements creation, they supervise the whole process, define domain concepts, further refine the metamodel and participate in reviewing the quality of the metamodel during the validation and testing process. Therefore, DSML development becomes more approachable for non-experts that would be otherwise deterred by the formality and complexity of the traditional language design approach.

Various names have been proposed for the top-down metamodel construction out of example models, among them ”Example-driven metamodel development”, ”Demonstration-based approach for domain-specific modeling language creation” or ”Bottom-up designing domain-specific modeling languages”. In order to establish a consent terminology, we propose the word *Moodling*, a blend between ”Doodling” and ”Modeling”. We will use this word for the remainder of the paper as an umbrella term for all DSML creation processes that fit into the process described the figure 1.3.

1.3 Tool overview

Several authors have recognized the need for more flexible and agile processes in MDE and identified example-based approaches as a possible alternative to established methods. Proposed techniques do not only target the metamodel design, but can be applied to other MDE artifacts as well. One area of research in MDE is concerned with model transformations, that is, the transformation between one or more source models to one or more target models by following a set of transformation rules [68]. As a consequence, research has been conducted that investigates methods to determine such transformation rules by example [88][3][4][48]. Other areas include model refactorings [32][31] and DSML validation [60]. Furthermore, the tool *Muddles* [50] extends the idea of automated metamodel inference by adding support for example models that can be processed by model management programs such as simulators, model-to-model and model-to-text transformations. This enables engineers to develop additional early confidence that the envisaged DSML fits its purpose.
Beside example-based approaches, there also exist tools that try to implement a more flexible metamodeling process by mitigating the conformance relationship between instance- and metamodel. While some of them rely on relaxing this relationship in the early phases of modeling [44][84][70], others let the metamodel evolve together with the instance models [34][36]. Lastly, it has been tried to make MDE more suitable for agile development processes by implementing team collaboration and multi-view features [21][30]. For example, *AToMPM* is a multi-paradigm modeling tool which elevates web-based technologies for a multi-user cloud-based architecture [86]. Other tools which provide development environments running in web browsers are *Clooca* [45], *WebGME* [64] and *MDEForge* [7]. However, all of these tools concentrate mainly on the modeling phase and therefore do not fit in the category of sketch-based metamodel construction.

An overview of the tools that focus on the actual DSML creation process is shown in table 1.1. *MLCBD*[1] (Modeling Language Creation By Example) provides a systematic and user-centered approach for building visual DSMLs. It is capable of inferring a metamodel and static constraints based on a set of domain example models and automatically generating instance models from the metamodel to assist the user in exploring the model space. *Scribbler*[2] focuses on the sketching aspect in the early design phases of model-driven engineering. As such, it aims to emphasize a collaborative and creative workflow where models can be drawn first as free-hand sketches and are then transformed to formal models while being independent from a pre-defined modeling language. A similar approach has been proposed and implemented in the *FlexiSketch*[3] tool. Its goal is to better integrate the early sketching and modeling activities in the overall software engineering process. This is achieved by enabling the user to sketch instance models without prior definition of a metamodel. Furthermore, a metamodel can be derived semi-automatically from these instance models. The tool was later extended with support for team collaboration. *metaBup*[4] aims to simplify the DSML construction process and make it more approachable for engineers without metamodeling expertise. It proposes an interactive,

[1] `http://hcho7.students.cs.ua.edu`
[2] `http://sse-world.de/index.php/forschung/ergebnisse-im-video/scribbler-from-collaborative-sketching-to-formal-domain-specific`
[3] `http://www.ifi.uzh.ch/en/rerg/research/flexiblemodeling/flexisketch.html`
[4] `http://jesusjlopezf.github.io/metaBup/index.html`

Name	Year	Implemented	Resources
MLCBD	2012-2013	Yes (MS Visio plugin)	[15][17][14]
Scribbler	2013	Yes (Standalone)	[5][6][89]
FlexiSketch	2012-2015	Yes (Android App)	[92][93][94][95]
metaBup	2012-2017	Yes (Eclipse plugin)	[57][59][56][46][83][58]
Model Workbench	2013-2014	Yes (Standalone)	[79][78][81][80]

Table 1.1: Overview of investigated tools

iterative and example-based approach with automatic modeling environment generation. It lets the user sketch example models by using informal drawing tools which are then used to infer the abstract and concrete syntax. Finally, the *Model Workbench*[5] has been used as a platform to implement a bottom-up metamodel design process based on example models. Early work on textual example models was extended to support arbitrary example models independent of a concrete syntax.

In the remainder of this paper, the presented Moodling process as well as the accompanying tools and approaches will be further evaluated. First, section 2 discusses the individual challenges of Moodling and shows how different solutions solve them. Section 3 evaluates the presented tools using various criteria and concrete test scenarios. This will help to point out gaps in the discussed approaches and to identify areas where further research is needed. Section 4 concludes the paper and gives an outlook on such possible future research.

[5]http://www.neu.uni-bayreuth.de/de/Uni_Bayreuth/Fakultaeten/1_Mathematik_Physik_und_Informatik/Fachgruppe_Informatik/Angewandte_Informatik_IV/de/research/projects/983_ModelWorkbench/index.html

2 Challenges

While the main challenge is to find an algorithm that takes the set of example models and generates a common metamodel, a holistic approach to Moodling also incorporates other challenges that will be discussed in this chapter. While numerous solutions were proposed for Moodling, they differ significantly depending on the respective focus of research. For example, one solution focuses on textual DSML development while another solution investigates the challenge of metamodel evolution within the developed framework. By discussing each of the identified challenges individually, we can perform a structured analysis of existing solutions, compare them and point out aspects that are still unsolved. Section 2.1 focuses on sketch recognition which is often used in Moodling to allow for free-hand input of the set of example models. Although this topic has been researched quite well, the reliable recognition of hand-drawn shapes and forms within metamodeling still poses a challenge. Section 2.2 describes how the concrete and abstract syntax are inferred from the example models. This forms the backbone of every approach since it is concerned with the problem of deriving a formal metamodel from a set of potentially incomplete, inconsistent or insufficient example models. Section 2.3 introduces the challenge of modeling languages evolution in general and the co-evolution of instance- and metamodels in particular. This is an important aspect since DSMLs typically undergo multiple iterations and evolve over time. To avoid inconsistencies, the conformance relationship between the derived metamodel and the example models has to be preserved when either the metamodel or the example models change. Section 2.4 deals with the issue of further utilization of the generated metamodel. After the inferred metamodel has reached all quality and domain requirements, it will be used as descriptive element of the DSML's structure and static semantics. Therefore, some solutions automatically generate a modeling environment for the created DSML, while others allow the export of the metamodel to common metamodeling frameworks. Lastly, section 2.5 looks at the tool support of the solutions. While some proposed solutions may only be of theoretical nature, others focus on the practical implementation of the approach to demonstrate its usefulness.

2.1 Unconstrained input

Unconstrained input is the first step towards the inference of metamodels from example models as one of the prerequisites for Moodling is the possibility to freely create and edit graphical sketches of the example models. In figure 1.3, hand-drawn shapes together with sketch recognition is the most commonly applied technique for step 1, namely the example model creation phase. Ideally, sketches are partial or complete instance models of the DSML to be designed and focus on a single specific aspect of the language only. Every sketch consists of a set of shapes which represent the concrete syntax for an element in the abstract syntax. Sketches have to be recognized and categorized in order to infer a set of distinct elements that form the concrete syntax. Furthermore, the relationship between the shapes has to be captured as well in order to reason about the abstract syntax. The automated recognition of hand-drawn shapes is called sketch recognition and has been researched exhaustively with the applications being multifold, ranging from the detection of hand-drawn chemical structures [74] to electric circuit design [28]. In the scope of systems modeling, it has been used for example to recognize hand-drawn UML shapes [38][11] or user interface design [52]. Generally, sketch recognition algorithms can be classified into gesture recognition (how the sketch was drawn) and visual recognition (what the sketch looks like), although combinations of both do exist as well [39]. Gesture recognition requires a shape to be drawn in a particular drawing style to achieve a high detection rate while visual recognition depends on an accurate geometrical description of the shape [20]. However, there is broad agreement that sketch recognition only works in domains where a well-established lexicon of possible shapes exists and that unrestricted recognition (the recognition of arbitrary shapes

without prior definition of a lexicon) is not feasible [25]. This poses an issue for Moodling where the goal is to give the user as much freedom as possible while sketching the example models of the DSML.
Depending on the approach, sketches are created either in an external general-purpose drawing software such as Dia[1], Microsoft Visio[2] and yEd[3] or directly within the Moodling environment itself. With the latter, the sketch recognition algorithm not only has access to the coordinates of the drawn data points but can also to the timestamp when these points were drawn, thus enabling gesture recognition. Some solutions use this additional information to increase the recognition rate. Due to the difficult nature of sketch recognition, the investigated approaches either support recognition of previously defined shapes or avoid the problem altogether.

1. *Recognition of pre-defined shapes:* In their Moodling tool *Scribbler*, Vogel et al. support sketch recognition of hand-drawn shapes within the environment itself [5][6][89]. It requires a lexicon of previously defined shapes that the algorithm should recognize. Therefore, Scribbler comes with a training feature where new shapes can be demonstrated and saved in the lexicon for later use. Both the learning and recognition method are based on a trainable sketch recognizer introduced by Coyette et al. [22]. It uses the Levensthein distance to compute the edit distance between the unknown shape and the shapes in the lexicon. The shape with the smallest edit distance is considered to be the type of the unknown shape. For every shape, *Scribber* saves the x- and y-coordinates, the convex hull and the mouse movements which were used to it. By using the convex hull, the shape is then scaled to a previously defined grid size. After that, the intersections of the drawing's lines and the grid itself are calculated. By enumerating the grid's individual rectangles, the algorithm calculates a set of numbers describing where and in what order the shape crosses the boarders of the grid. This set can then be compared using the Levenshtein distance with the already known elements in the lexicon. To add a new element to the lexicon, *Scribbler* has a dialog where the user can repeatedly draw the desired shape to populate the lexicon with more samples.
   A similar approach is used in the (MLCBD) framework [15] by Cho et al.: It supports both the recognition of a set of default shapes and a so-called shape authoring tool to register domain-specific shapes. However, it uses not only visual, but also gesture recognition to make the process more robust. The sketch recognition is based on *SUMLOW*, a tool for recognizing UML diagrams sketched on an electronic whiteboard [12]. *SUMLOW* features both a multi-stroke recognizer for UML shapes and a text recognizer for annotations of the UML elements. The multi-stroke recognizer makes use of a database that describes inherent features of the various UML shapes and how they are typically drawn. It therefore not only captures the shape itself but also records the strokes that were executed to draw it. As a result, it can only detect shapes when they are drawn in a specific order (for example, the *Actor* shape needs to be drawn with the head as a circle first), but with very high detection rate. As soon as a UML shape is recognized, it is automatically extended with empty text fields for the annotations. When the user writes inside the text fields, a separate text recognition transforms the handwriting in text. MLCBD uses the same technique as it requires the user to register new shapes first. However, it also defines a set of pre-defined shapes that cover commonly used shapes such as circles, rectangles and arrows. The main disadvantage of this approach is that the shapes can only be recognized if they are drawn in a specific order. Therefore, the user has to be familiar with the particular shapes and stroke drawing process.
   Finally, also the *FlexiSketch* framework by Wüest et al. uses a lexicon of shapes to recognize via Levenshtein distance comparison [92]. However, the lexicon is populated in realtime while the user draws shapes and therefore transparent: if an object is not recognized, the user is asked to provide a name for it so it can be inserted in the lexicon. If a sketch is recognized since it is already in the lexicon, it is added to the set of sketches representing the sketch to further train the recognizer.

2. *No recognition:* Due to the inherently difficult nature of sketch recognition, the *metaBup* framework [56][57] avoids this problem altogether by letting the user manually provide a legend for the shapes along with the sketched example model (also called "fragment" in the framework). This legend maps every every shape in the example model to a metamodel type. This also allows using different symbols for the same concepts. The example models are not drawn in the tool itself but externally by using general-purpose drawing tools. So far, the importer of *metaBup* supports the file formats of the drawing tools *yEd* and *Dia*. This is made possible by a common sketch metamodel that imposes a specific structure on the input sketches. It captures most of the types typically found in sketches, including nodes, lines, arrows, text labels and graphical properties of the aforementioned. A *metaBup* example model with a legend is depicted in figure 2.1. This simple fragment serves as an example model for a DSML that allows the user to describe the connection between an internet service provider and customer homes. Apart from the example model, the legend assigns every icon a name that will later be used as type in the metamodel. This approach bears similarities to the traditional top-down DSML construction process where the mapping between abstract and concrete syntax is often carried out manually. Typically, some kind of icon definition languages are used where the user has to explicitly create an icon for each meta-class [87].
   Later, the framework was extended by more advanced detection techniques mainly for spatial relationships such as contain-

[1]`https://wiki.gnome.org/Apps/Dia`
[2]`https://products.office.com/visio`
[3]`http://www.yworks.com/products/yed`

ment and overlapping shapes [59]. This allows for a more detailed representation of domain properties. Recognized spatial relationships are translated to annotations for the textual model representation and later to references in the generated metamodel.

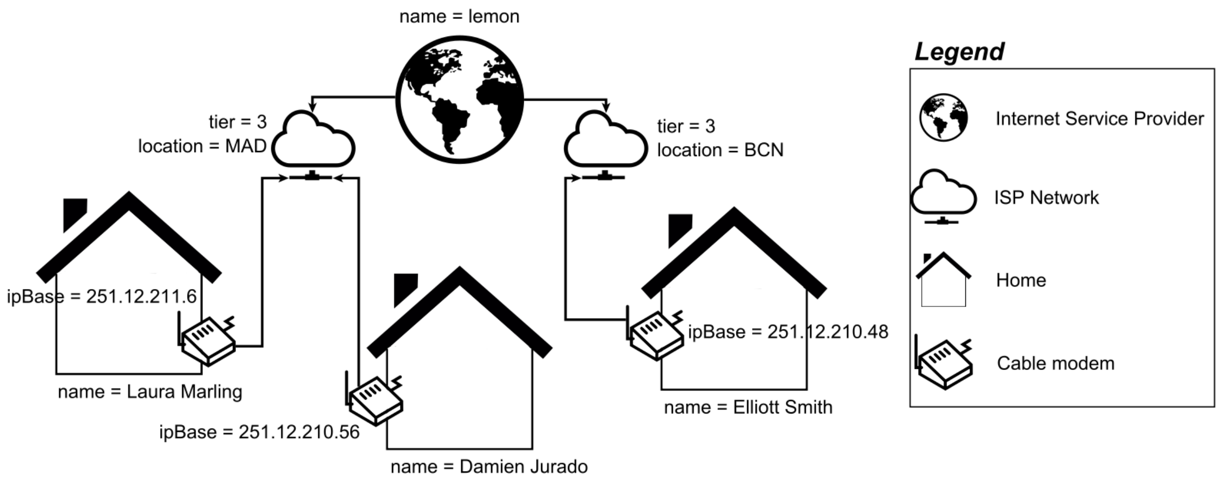


Figure 2.1: Example of a *metaBup* input fragment consisting of sketch and legend

The majority of the investigated approaches above focus on the graphical development of DSMLs and therefore require some kind of sketch recognition or equivalent. However, Roth et al. describe in their work a bottom-up development process that focuses on textual DSMLs [80]. Consequently, the example models are of textual nature and do require some kind of text parsing and recognition. This is done with a mixture of regular expressions and manual user input, as further described in section 2.2. Moreover, Roth et al. identify a need for features such as syntax highlighting and content assist since they are common for modern IDE and expected by the user.

2.2 Metamodel generation

The metamodel captures the abstract syntax and the static semantics of a DSML and constrains the structure of the language elements. Therefore, probably the most important challenge in Moodling is the inference of the metamodel from the set of example models. Ideally, an inference engine can use a large set of concise and consistent input models. In reality however, the input models might be incomplete, inconsistent or not reflect all aspects of the language appropriately. Therefore, the inference engine must be able to handle all of these potential issues.
Metamodel inference is also used in areas outside of Moodling. For example, *MARS* [47] is a system that tries to recover a lost metamodel from domain models by using a semi-automatic grammar inference engine. It induces a metamodel by first translating the set of domain models into a custom model representation language which then is used to translate the domain models into a context-free grammar. From this grammar, the metamodel is generated. Although being rather constrained to the specific environment and format of the domain models, it shows promising results in an experimental study. However, in this approach, the lost metamodel has already been defined in the past and the set of domain models can therefore be considered consistent and complete.
We classify the metamodel inference capabilities by their level of automation. While some approaches are able to infer the metamodel in a completely automated manner, others require user interaction or are very limited in that the metamodel is essentially generated by the user itself.

1. *Manual: Scribbler* requires the user to externally define the metamodel of the DSML within the Eclipse Modeling Framework (EMF) as an Ecore model [5]. The user then has to manually map the previously defined sketch elements of the language to the metamodel objects. After this, the user can sketch instance models that can be exported to the EMF for further usage. This mapping is bidirectional since *Scribbler* also supports the import of EMF models which then get represented as sketches again [6]. Therefore, *Scribbler* is well suited for language experts that are able to define the metamodel of the DSML within EMF but rather like to sketch their models by hand than a complete Moodling solution where the metamodel is inferred from a set of sketched instance models.

2. *Semi-automatic:* MLCBD supports the semi-automatic inference of the DSML metamodel. Proceeding from the recognized sketches (see section 2.1), it defines an algorithm that incrementally builds the metamodel with user interaction where necessary [17]: First, the so-called Graph Builder transforms the sketched example models into representation-independent

undirected graphs by using graph transformations. The Concrete Syntax Identifier then traverses the generated graphs and tries to identify candidates for the concrete syntax of an element. The candidates need to be reviewed and annotated by the user during this step, thus making the approach semi-automatic. The annotations help to further specify the nature of the elements (e.g. does a link denote inheritance, is it bidirectional or unidirectional). After the concrete syntax has been identified, the Graph Annotator takes the previously made annotations and applies them to the graph representations of the example models generated in the first step of the process. It renames the elements from their generic to the concrete syntax names, changes the graphs to directed graphs if necessary and optimizes the graph structures by merging nodes with the same names and attributes. The optimized graph structures are handed over to the actual Metamodel Inference Engine that infers the abstract syntax and outputs the metamodel of the input models. This is done by first merging the graph representations into one single graph that represents all aspects of the input models and then performing a (sub)graph isomorphism test between instances of known metamodel design patterns and the merged graph. These design patterns are a result of a survey conducted by the same authors of MLCBD [16]. In his PhD thesis, Cho extended the described approach with model space exploration [15]: After the metamodel is successfully inferred, a set of example models is automatically instantiated from it. The user then plays the role of an oracle and decides if the generated model is valid in the DSML. This information is then fed back into the Inference Engine to incrementally update and further refine the metamodel. This approach also helps to identify the current quality of the induced metamodel and possibly spot mistakes that otherwise would only appear later when the DSML is used in production.

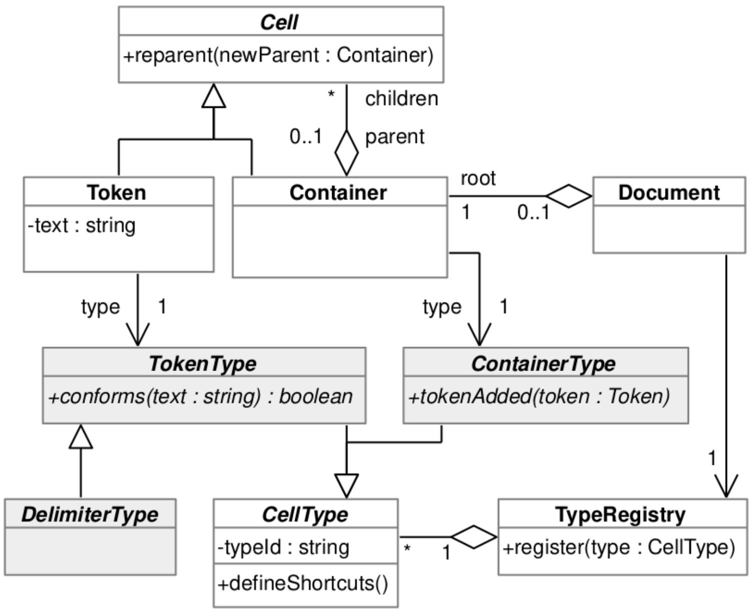
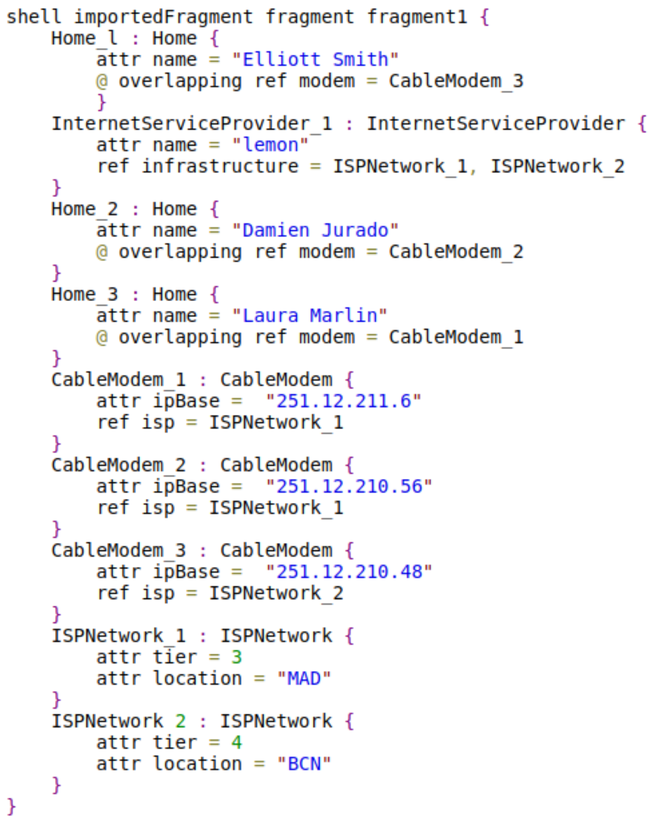


Figure 2.2: The metamodel for the concrete syntax tree according to Roth [80]

Another approach was described by Roth et al. [78][80][81]. Unlike the other discussed solutions, it focuses on Moodling where the input models and the generated metamodel are of textual nature only. Similar to abstract syntax trees, the solution proposes the use of a concrete syntax tree (CST) that is built gradually while the user enters a textual example model. It also provides a metamodel that describes the structure of such a CST which is depicted in figure 2.2. A base class called "Cell" serves as an abstract class for the tokens and containers classes. Both tokens and containers have an associated "TypeClass" that determines their type. To effectively support token recognition, several assumptions are made about the format and the semantics of the tokens. For example, a whitespace character is assumed to be a delimiter, a list of comma-separated tokens is an enumeration and braces or indentation are used to set the scope. These tokens are building blocks for containers which can be statements, complete blocks or expressions. The recognition itself is performed semi-automatic with regular expressions and manual user interaction. Regular expressions are able to identify single tokens and distinguish between literals such as integer and floats, but not between different types of tokens such as keywords, identifiers or references. In these ambiguous cases, user interaction is needed to determine the type of a specific token and register it in the CST. Once the CST is built, it can be used to derive the abstract syntax for the DSML as described in a seperate paper that concentrates on a method to derive a concise metamodel from example models [79]. In contrast to other approaches that focus on sketched input models, the described solution is independent from the concrete syntax of the example models. Similar to MLCBD [15], it supports three types of refactorings that are applied to the generated metamodel to reduce its complexity: Single inheritance, multiple inheritance and enumeration. The inference process itself only requires a notion of concepts, assignments and attributes that in principal can come from any previously executed recognition process such as the CST that was described earlier. The inference is then done in 4 steps: First, for every identified unique type, a new meta concept is created. Second, for each assignment in the type bodies, a new attribute is created for the corresponding meta concept. This can be done in a straight-forward fashion for literals such as strings or integers but is more complex for references to other meta concepts since the cardinality has to be taken into account. As a general rule, the lower bound is set to 1 if each instance of the same type contains such a reference, otherwise 0. The upper bound is set to 1 if only one value is assigned every time, otherwise to *. In the third step, duplicate attributes are merged. Finally, the fourth step optimizes the metamodel by introducting inheritance and enumerations.



```
shell importedFragment fragment fragment1 {
    Home_1 : Home {
        attr name = "Elliott Smith"
        @ overlapping ref modem = CableModem_3
        }
    InternetServiceProvider_1 : InternetServiceProvider {
        attr name = "lemon"
        ref infrastructure = ISPNetwork_1, ISPNetwork_2
    }
    Home_2 : Home {
        attr name = "Damien Jurado"
        @ overlapping ref modem = CableModem_2
    }
    Home_3 : Home {
        attr name = "Laura Marlin"
        @ overlapping ref modem = CableModem_1
    }
    CableModem_1 : CableModem {
        attr ipBase =  "251.12.211.6"
        ref isp = ISPNetwork_1
    }
    CableModem_2 : CableModem {
        attr ipBase =  "251.12.210.56"
        ref isp = ISPNetwork_1
    }
    CableModem_3 : CableModem {
        attr ipBase =  "251.12.210.48"
        ref isp = ISPNetwork_2
    }
    ISPNetwork_1 : ISPNetwork {
        attr tier = 3
        attr location = "MAD"
    }
    ISPNetwork 2 : ISPNetwork {
        attr tier = 4
        attr location = "BCN"
    }
}
```

Figure 2.3: *metaBup* text fragment, automatically generated from an example model

3. *Automatic: metaBup* implements a fully automated metamodel inference engine [57][56]. Starting from the sketches, an importer generates a textual fragment that describe the classes and references between the classes in the sketch. This importer first converts the sketch to a model that conforms to the sketch metamodel of *metaBup* (see section 2.1). It then uses a model transformation to transform the model into a text fragment. This process is necessary to stay technology agnostic when different sketching tools like *Dia* or *yED* are used. To effectively support a model transformation from the sketch to the text fragment, *metaBup* also features a metamodel for the text fragments language. A text fragment consists of objects with attributes and references with both of them being able to be annotated. The elements of the sketch metamodel are mapped to the elements of the fragment metamodel to carry out the model transformation. Figure 2.3 shows such a fragment that was automatically generated from the sketched example model in figure 2.1. Particularly, the spatial information about overlapping icons that is present in the example model got translated to textual annotations in the fragment.
After the transformation, the metamodel inference engine iterates through all text fragments and creates a new meta-class for every object found, if it does not exist already. Similarly, an annotation is created for every reference found in the fragments. The cardinalities are set to the respective minimum and maximum numbers as found in the fragment. The resulting metamodel can then be refactored and evolved as described in section 2.3. To guide the metamodel inference process, the user can also manually annotate either the sketches or the text fragments. Such annotations are either domain or design annotations: domain annotations give the inference engine knowledge about further domain knowledge and result in OCL constraints attached to the generated metamodel (see section 2.4) while the design annotations affect the structure and organization of the metamodel. However, these annotations are optional and the inference process can be executed from the sketches to the final metamodel in a completely automatic manner.

Automatic metamodel inference is also supported by the *FlexiSketch* tool [92][93]. Users can draw symbols, links and annotations on a canvas and are required to provide type information and a name for every shape. For symbols, the user can choose to either use already registered types or add the symbol as a new type. Therefore, the set of available metamodel classes typically grows during the sketching process. For links, the user has to select the type from a set of given options such as unidirectional, bidirectional, solid or dashed. This limits the amount of possible link types and permits symbol overloading. Lastly, the cardinalities are automatically set and updated when needed. However, the user can also select a link and set the lower and upper bounds explicitly as long as they do not violate any other cardinality rules. Finally, to ensure consistency of the sketched model, a wizard can be used during sketching or when the model is saved to fix issues such as missing type information for symbols and links.

2.3 Evolution of modeling languages

Software evolution is a subdomain of software engineering that investigates ways to adapt software to changing requirements and operating environments as well as the change process itself [67]. Due to the general-purpose nature of most programming languages, only the programs evolve, while the languages themselves do not. This is contrary to MDE where modeling languages are often tighly bound to a specific domain. Therefore, both the instance models and the language itself are subject to frequent changes due to changing requirements or refactorings. Due to the importance of the topic, the MDE community has conducted intensive research to find solutions for what is often referred to as the *co-evolution* of models [42]. In their recent survey, Hebig et al. give an overview of 31 different approaches that investigate metamodel co-evolution [43]. Meyers and Vangheluwe present a taxonomy for different evolution scenarios within MDE and propose a structured, semi-automatic framework for the evolution of modeling languages [69].
Cho et al. identify the co-evolution of models as one of the key challenges in Moodling [18]: Language evolution becomes particularly interesting because only rarely, all language requirements are met or even known within the first development iteration. In Moodling, changes to the language are either induced by adapting the set of example models and then propagating the changes to the language induction mechanism (which we will call *forward evolution*) or directly manipulating the derived metamodel (*backward evolution*). Both cases carry the substantial risk of instance models and metamodel diverging and therefore breaking the conformance relationship between example models and metamodel. Furthermore, a change to one single example model might affect all other example models as well, making the evolution problem even more of a challenge. For this reason, many solutions identify a need for a (semi-)automated co-evolution of models. This paragraph discusses the different approaches to model co-evolution found in the literature. We will use the term *unidirectional* to describe that co-evolution is only supported in one direction (that is, either forward or backward evolution but not both). The term *bidirectional* denotes that both forward and backward evolution is supported.

1. *Unidirectional, manual:* Cho identifies language evolution as the weakest area of the MLCBD approach [15]. He compares MLCBD with GME, a generic modeling environment focusing on easy creation of graphical DSMLs in the traditional top-down process [24] and concludes that while GME supports evolution of the metamodel by applying requested changes and triggering a rebuild of the metamodel, MLCBD requires a complete iteration of the development process. Therefore, the only possibility to evolve the language is to change the set of example models and perform a full iteration of the process again so a new metamodel is created which reflects the changes. Also, no refactoring or backwards propagation of manual changes from the metamodel to the example models seems to exist within MLCBD.
   The collaborative model sketching approach *Scribbler* implements manual backwards evolution. Although not explicitly mentioned in any of the resources, it can be deduced that the metamodel, since it has to be created by hand, can also be changed at any point of the DSML development process. However, it is also the user's responsibility to manually maintain a valid mapping between the metamodel elements and the sketches. Forward evolution is not possible since the workflow needs a metamodel defined prior to any sketching activities.

2. *Bidirectional, semi-automatic:* In their effort to create an interactive and iterative Moodling approach, *metaBup* by López-Fernández et al. support the refactoring of the metamodel after it is automatically induced from the example models [57][56]. They implement a visual refactoring assistant that provides several pre-defined metamodel improvement suggestions to the user, ranging from simple renames to pulling up common features to a common superclass. These refactorings stem from those found in traditional object-oriented programming as exhaustively described by Fowler [29] and Demeyer et al. [26]. In this example for backward evolution, they follow the taxonomy described by Cicchetti et al. [19]: Metamodel changes are classified into *non-breaking, breaking and resolvable,* and *breaking and unresolvable*. The solution supports automatic updating of the example models if the metamodel refactoring is either non-breaking or resolvable. For the unresolvable ones, the user is asked to provide additional information or simply discard example models that no longer conform to the updated metamodel. The solution also allows for forward evolution where the metamodel is changed by changing the set of example models. If an example model is altered in a way that it now includes contradictory information, the assistant raises a conflict and notifies the user about the issue. If the conflict is resolvable, it tries to automatically resolve

the issue. For instance, changing an attribute to a string while another object already defines the same attribute as an integer results in an automatically resolvable issue since all integers can be represented as strings as well. However, the other way around presents an unresolvable conflict that requires manual user intervention since not all strings can be represented by integers.

3. *Unidirectional, automatic: FlexiSketch* tries to support automatic forward co-evolution of the example- and metamodel [93]. Two mechanisms are presented to minimize the synchronization problem between multiple versions of the models. First, each sketched example model stores its own metamodel that it conforms to. At any point, metamodels can be merged automatically as long as only non-breaking changes such as relaxing existing cardinality rules need to be performed. Merges that result in breaking changes are rejected and left for the user to resolve manually. Second, metamodels can be locked as soon as they are considered final. Once a metamodel is locked, it will not be updated anymore even if the example models change. Instead, the parts that do not conform to the metamodel are highlighted. A locked metamodel can also be saved independently from the example models.

4. *Bidirectional, automatic:* In his dissertation [78], Roth extends his previous work on textual Moodling [80][79] and describes a holistic approach where he distinguishes between the linguistic and the onthological context. Both refactorings for the generated metamodel and induced changes from the example models are supported in a fully automated way. This is made possible by restricting the changes that can be applied to either the metamodel or the example models. For the refactorings, only changes that are strictly non-breaking and do not require any changes to the models conforming to the metamodel are performed. This includes for example further restriction of multiplicities and the deletion of unused concepts. The same holds true for forward evolution: The user can freely manipulate the set of example models and regenerate the metamodel as long as the changes do not invalidate the conformance of any model with the metamodel. In such a case, an issue is raised and the user is asked to resolve the problem manually.

2.4 Modeling environment integration

The main goal of Moodling is the creation of a conceptual metamodel for a DSML from instance models. However, once this metamodel has been inferred, it still needs to be implemented in a particular platform to serve as a usable description for a DSML. We distinguish the solutions between standalone applications that feature an export function to established modeling environments such as the Eclipse Modeling Framework (EMF)[4], *metaDepth*[5] and *MetaEdit+*[6] or solutions that are already integrated into such environments and therefore do not require to switch the tool during the DSML development lifecycle.

1. *Export:* For standalone applications that implement a Moodling approach, it is common to export the generated metamodel to an already existing framework. Ideally, the existing framework is well-known and mature so a high quality environment is ensured. As already mentioned in section 2.2, *Scribbler* is a standalone tool, but tightly integrated with the EMF as it requires a metamodel definition as Ecore metamodel to map recognized sketches to classes. Once a sketch is done, it can be exported to the EMF. *Scribbler* therefore allows for sketching instance models to already defined DSMLs. It also supports the import from EMF models to represent them as sketches again, given the mapping between sketch elements and Ecore classes is still intact [5].
   FlexiSketch also relies on export. The developers plan to integrate an export function for the generated metamodel to MetaEdit+, another environment for creating and using DSMLs [93]. Currently, however, it only supports saving and loading a metamodel in a custom XML format. It is suggested that this XML file can be manually converted by the user to the format of the target environment until an export function is available.
   Although *metaBup* is implemented as a plugin to the EMF and therefore primarily advertises an automated modeling evironment generation within the same, it also supports exporting its metamodels to metaDepth [57]. Hereby, the exporter makes use of features that are unique to metaDepth, for example the distinction between unidirectional relations between classes called ”Reference” and bidirectional relations called ”Edge”. Constructs that cannot be directly expressed in metaDepth metamodels are compiled into OCL constraints.

2. *Integrated:* As mentioned before, *metaBup* is implemented as a plugin for the Eclipse Modeling Framework (EMF) [57][56]. Although it supports the export of the generated metamodel to metaDepth, it mainly focuses on the automatic generation of a modeling environment for the DSML within the EMF [59]. This includes the transformation from the neutral metamodel to a specific Ecore metamodel that defines the abstract syntax with the concepts provided by the EMF, a model for the concrete syntax for Sirius, the language workbench creation software of the EMF, and a mapping between those two. These transformations are possible due to the fact that *metaBup* provides an explicit metamodel for the example models. Therefore, a mapping between elements of the example models and the EMF Ecore metamodel can be established.

[4]`https://www.eclipse.org/modeling/emf`
[5]`http://metadepth.org`
[6]`http://www.metacase.com/products.html`

The *Model Workbench* was extended by Roth to support his Moodling approach [78]. It was initially developed to support DSML development for the business process domain. Both textual and visual metamodel development methods are supported. Once the metamodel is defined, the framework allows to instantiate models from it within the same environment by switching from the DSML design view to the modeling view. For visual DSMLs, it provides the user with a pallet of all possible constructs in the language and notifies about metamodel violations in real time. The textual editor supports syntax highlighting and auto-completion of language keywords.

2.5 Tool support

The tool support of an approach can be used as a benchmark for its practicability. While some implement a complete production-ready software for Moodling, others only provide a prototypical implementation to prove the general feasibility of the idea. This paragraph describes the tools that were already presented in chapter 1 and categorizes them into either prototype or fully functional software.

1. *Prototype: metaBup* is implemented as a technical prototype to show the feasibility of the approach described by López-Fernández [57][56]. It is built as a plugin for EMF and was evaluated in a user study to investigate the usability and quality of the tool [59]. For this, 11 participants were asked to create sample models from a pre-defined domain and set of symbols. Although the results show that both the usability and quality of the tool and the generated modeling environment are high, the narrowness of the study permits a statement about the universal quality of the overall approach for unconstrained Moodling.
   Instead of implementing the solution in an already existing modeling framework, MLCBD emphasizes the sketching aspect of the approach by realizing a prototype in the general-purpose graphics application *Microsoft Visio* [15] which allows plugin development and process automation with Visual Basic for Applications (VBA). Two case studies were conducted by the author where MLCBD is used to develop two different DSMLs and compared to other modeling environments that only support the traditional top-down, metamodel-centric design process. The author concludes that with MLCBD, less effort is required to develop a DSML since no language development expertise is needed to create the language. However, he also identifies limitations when it comes to language evolution (see section 2.3) and the flexibility of the VBA implementation within MS Visio.
   FlexiSketch is implemented as a prototype application for mobile devices powered by Android [92]. Thereby, it focuses mainly on devices with big screens, i.e. tablets for proper sketching support. A screenshot of the tool can be seen in figure 2.5 and shows how different features like free-hand sketching, linking and element editing can be done on the same canvas. The tool was evaluated in two different studies conducted by the authors. While the first study focused on the general feasibility and usability, the second study was concerned with the utility of the tool. In conclusion, both studies were rather small but showed promising results: most participants only had critique on details and the vast majority rated the tool as useful and practical for their daily work. The single most frequently mentioned improvement suggestion was support for larger screens and electronic whiteboards. Recent developments of the FlexiSketch tool address these issues by implementing a client-server architecture that supports a broader range of devices and also team collaboration [94].

2. *Production-ready: Scribbler* is a fully functional and collaborative Java application to transform free-hand sketches to formal models and back again [5][6]. It focuses on the collaboration and sketching aspect and, although being an independent tool, is closely integrated with the EMF and its Ecore metamodel for the abstract syntax and semantics. To support collaboration, *Scribbler* is implemented as a client-server architecture and transferring the sketches to all clients simultaneously. It also features a history function that saves all drawing actions and lets the user replay the sketching process later on. The tool was evaluated by three different industrial users that all confirmed the usefulness during the early phases of development. Especially the collaboration functionalities and the high recognition rate of new sketches were perceived as valuable. Figure 2.4 shows a screenshot of the tool where the user has drawn an example model from a causal-block diagram DSML [35] with operator and constants blocks and connections between them.

   Roth [78] implements his approach in a web application called *Model Workbench* that was initially developed for modeling domain-specific process management modeling languages. It comes with two different editors for textual and graphical metamodeling. Following his previous work [80][79][81], the approach focuses on textual Moodling. As described in section 2.2, a token parser is implemented that semi-automatically constructs a concrete syntax tree (CST) from textual example models. This CST is then used to infer a metamodel which can be reused within the same environment. Once the metamodel is derived, the instance model editor supports syntax highlighting and autocomplete of the previously identified keywords.

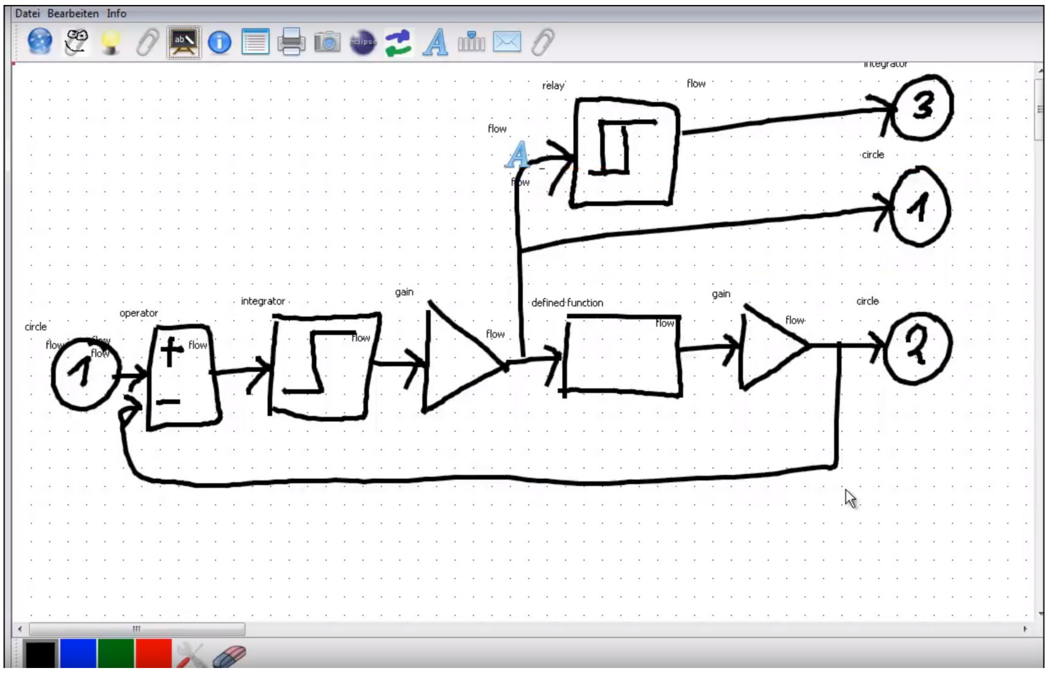


Figure 2.4: Screenshot of the *Scribbler* tool

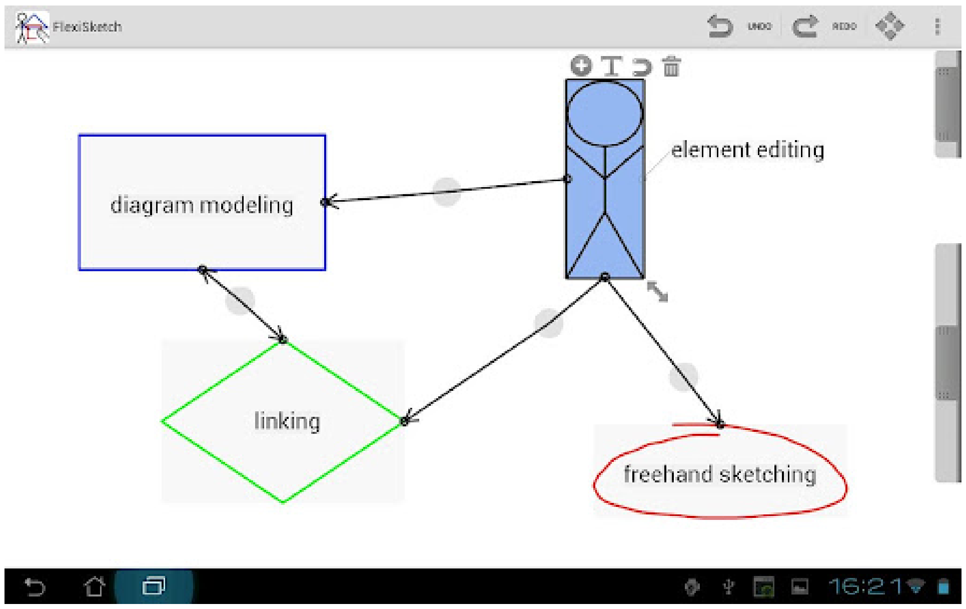


Figure 2.5: Screenshot of *FlexiSketch*

3 Analysis

In this chapter, the quality of each solution will be analyzed with respect to the challenges presented in section 2. In order to establish a common ground for the analysis, concrete test scenarios are developed for every of the challenges. These scenarios serve as a benchmark framework for each tool and make a quantitative comparison between them possible. Each following section first introduces a test scenario that reflects a individual challenge and then investigates if and how it manifests in the different tools. It should be noted however that, since most tools were not available for an actual hands-on evaluation, most of the analysis was carried out based on the information found in the literature.

3.1 Unconstrained input

This section assesses the quality of the input method for example models of each tool. This both includes what kind of visual building blocks and variables are available to express the user's intent and how the tool uses this information to infer a metamodel. Such visual variables include, but are not limited to the shape, color, textual annotation and spatial relationship of a symbol [72]. Although rectangular shapes are the most predominantly used notation in visual languages [76], it can not be anticipated what constructs the user will use for the example models. Therefore, a good the input method provides a wide variety of visual notations and is able to recognize and differentiate between them. In order to limit the amount of test cases needed analyze the input methods, only constructs that are explicitly supported by any of the tools are used. This includes the three variables shape, textual annotations and spatial relationships. For each of these variables, one example model was developed that assesses these variables individually. Since the concrete representation of such example models will differ between the tools, an external drawing tool not affiliated to any of the tools was used to present the models in a neutral way.

An example model for the shape variable is shown in figure 3.1. It consists of named entities representing network devices and connectors as the links between them. No textual annotations or spatial relationships like intersections or containments are present. Therefore, the tool only has to provide means to input and recognize different shapes and connections. By adding annotations to the entities and typing the connectors as shown in figure 3.2, the example model becomes graphically more complex. The tool has to be able to not only recognize different shapes, but must also support textual annotations. Furthermore, there are now two different types of links present: a solid line which denotes an Ethernet connection and a dashed line for wireless connections. Finally, the last and most complex example model in figure 3.3 adds spatial information to the sketch. Network devices can now reside in different rooms (containment) and the router has overlapping circles representing ports. Consequently, the tool must possess a notion of layers so that shapes can overlap and contain each other.

1. *Scribbler* allows for unconstrained input by providing a canvas that lets the user freely draw shapes by hand. These shapes are then recognized by a sketch recognition engine, given that they are already registered in the lexicon of available shapes. To register new shapes, *Scribbler* provides a recognition training dialog where the user can repeatedly draw the shape of an element that should be recognized. The recognition process itself is done by computing the minimal Levenshtein distance between the drawn shape on the canvas and the shapes registered in the lexicon. This suggests that with *Scribbler*, it is possible to draw the basic example model of figure 3.1. However, no evidence could be found that implies support for more advanced constructs like annotations or spatial relationships as shown in the figures 3.2 and 3.3, respectively.

2. *MLCBD* is capable of both sketch recognition of hand-drawn shapes and modeling with pre-defined icons. It comes with

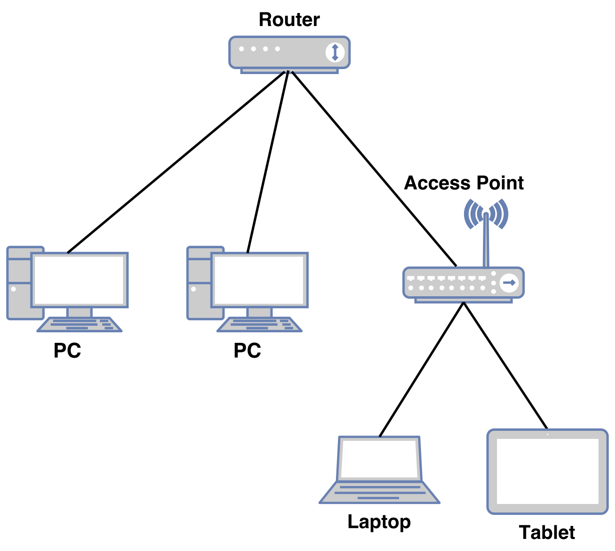


Figure 3.1: The most basic example model with entities and relations

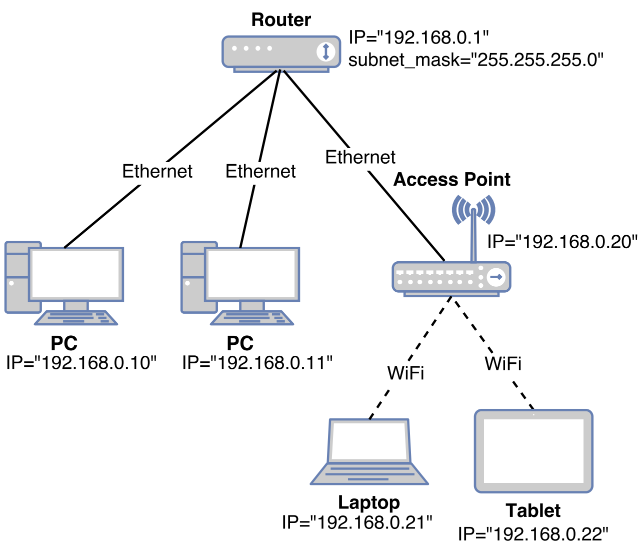


Figure 3.2: More advanced example model with annotations

an authoring tool to register new shapes that should be recognized later. Contrary to other solutions however, it not only records the coordinates of the strokes, but also the time (i.e., how a shape was drawn). This improves recognition rate at the cost that the user has to remember the individual steps that are needed to draw a specific shape. Given the fact that the tool is implemented as a plugin for *Microsoft Visio*, a commercial diagram and vector graphics application, the user is not forced to stick to hand-drawn shapes only, but can also use icons similar to the ones that are present in the test example models. Therefore, it can be assumed that example models with the complexity of the model in figure 3.3 can be sketched. However, the tool does not support the actual detection and processing of spatial relationships.

3. *FlexiSketch* tries to mimic the traditional pen and paper scribbling by supporting large-screen touch devices. While also featuring sketch recognition of shapes in a lexicon, it does not require an explicit learning phase: as soon as a drawn shape is not recognized, the user is asked to either manually pick a shape type from the lexicon or introduce a new type for the shape. Shapes can be annotated as shown in figure 3.2. However, such annotations are purely visual information and have no affinity to any shapes. Furthermore, no spatial relationships are supported, thus making it impossible to input any example models such as the one depicted in figure 3.3.

4. *metaBup* relies on external general-purpose drawing tools for the sketching process. It features an importer that can read the file formats of these external tools and transform them into an internal representation of example models. To map the elements in the sketch to metamodel objects, the importer also requires a legend of shapes that connects every shape with a type identifier. Therefore, the input is unconstrained but more laborious due to the fact that external tools need to be used for the sketching process. On the other hand however, the user is not constrained to one tool since the importer supports multiple of them, the most advanced being *yED*[1]. Since this editor comes with a set of pre-defined icons for various domains and also supports the import of custom icons, it is possible to sketch example models with the complexity of 3.3. The tool supports the detection of spatial relationships and translates them to metamodel constructs such as compositions and inheritance.

5. The *Model Workbench* implements a bottom-up DSML creation process for mainly textual languages. It provides an IDE-like text editor which allows the user to freely enter example models. Every key stroke triggers an analysis of the tokens, their types and the structure of the entered text. The result of this analysis is a concrete syntax tree that describes the structure of the example model. While later work suggests that support for visual languages has been added [78], the focus still lies on textual languages. As such, the test example models would have to be translated to a textual representation first. In that form, both attributes and spatial relationships can be represented, making the Model Workbench suitable for complex example models such as the one shown in figure 3.3.

The results of the analysis of the unconstrained input are summarized in table 3.1. *metaBup* and the *Model Workbench* support all test cases with their regarding graphical challenges. The *Model Workbench* primarily supports textual example models, where constructs like attributes and hierarchy can easily be expressed. *Scribbler* and *FlexiSketch* allow freehand sketching on a canvas

[1]`https://www.yworks.com/products/yed`

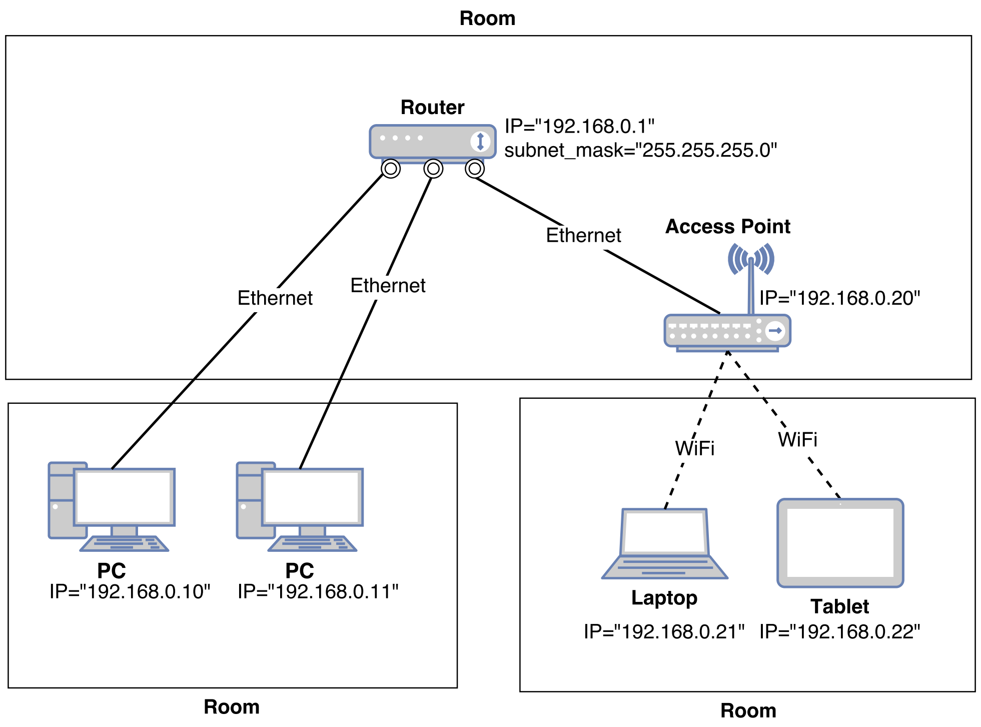


Figure 3.3: Complex example model with spatial relationships

object whereas *MLCBD* and *metaBup* make use of general-purpose vector graphics programs. While freehand sketching might be preferable due to being closer to natural pen-and-paper doodling, it is inherently limited in its power to express complex visual notations that go beyond basic shapes such as rectangles, circles and lines [72]. Furthermore, for the sketch recognition algorithm to work effectively, hand-drawn shapes need to be sufficiently distinguishable from each other, a constrain that is not present when using an icon-based language of annotated vector graphics as done by for example *metaBup*.

Tool	**Language type**	**Input**	**Entities and relationships**	**Textual annotations**	**Spatial relationships**
Scribber	Visual	Freehand	Yes	No	No
MLCBD	Visual	Editor	Yes	Yes	Yes (draw only)
FlexiSketch	Visual	Freehand	Yes	No	No
metaBup	Visual	Editor	Yes	Yes	Yes
Model Workbench	Textual and visual	Text editor	Yes	Yes	Yes (textual)

Table 3.1: Tool support regarding example model input capabilities

3.2 Metamodel generation

After defining a set of example models, a metamodel needs to be inferred from them. To evaluate the quality of this feature, the same example models that were presented in paragraph 3.1 are used. Obviously, the overall complexity of the generated metamodel is directly linked to the expressiveness of the example model: if a tool only supports shapes and links as depicted in example model 3.1, the resulting metamodel will only contain those entities. Another aspect to the analysis is the support for modeling concepts like inheritance, abstract classes or compositions which have a direct influence on the expressiveness and quality of the metamodel. For example, the metamodel for figure 3.2 could include an abstract class for the link, from which the two types "Ethernet" and "WiFi" are inherited as concrete classes. Finally, the level of automation is taken into account as well, although a preliminary classification of the tools has already been done in section 2.2. Again, due to the unavailability of most tools, this analysis has to be performed based on the information given in the literature.

1. Although being a standalone application, *Scribbler* is tightly integrated with the Eclipse Modeling Framework. As described in section 2.2, it requires the manual definition of an Ecore metamodel in Eclipse and a mapping between elements in the shape lexicon and classes of the metamodel. Therefore, no bottom-up metamodel inference is supported.

Tool	Metamodel generation	Advanced constructs	Automation
Scribber	Manual, external	None	Manual
MLCBD	Implicit	None	Semi
FlexiSketch	Implicit	None	Full
metaBup	Explicit	Inheritance, abstract classes, compositions	Full
Model Workbench	Explicit	Inheritance, abstract classes	Semi

Table 3.2: Tool support regarding the metamodel inference process

2. *MLCBD* proposes a semi-automatic bottom-up process. It makes heavy use of graph transformations to iteratively translate the input sketches to annotated graphs which are then used to infer the metamodel. Therefore, this approach allows for structural optimizations such as nodes merging. *MLCBD* uses a unique approach to infer the metamodel: After all example models are merged together to one graph, an isomorphism test is performed with known metamodel design patterns to identify common constructs and transform them into a metamodel for the DSML. However, the metamodel cannot be accessed or modified by the user and only exists in the form of an internal graph structure. Once a metamodel and its constraints have been inferred, the user can use the application to further create instance models that conform to the metamodel which was described by means of example models.

3. *FlexiSketch* combines the freehand sketching aspects of *Scribbler* with the hidden metamodel construction of *MLCBD*. While example models are sketched, the tool builds a metamodel which later is used for conformance checking new models that are sketched in the same language. The metamodel is incrementally built in the background while the user sketches. However, it does advanced constructs such as inheritance, abstract classes or compositions.

4. *metaBup* is capable of automatically deriving an explicit metamodel which is transparent to the user and can be manipulated. Since it supports detection of spatial relationships, a metamodel derived from example model 3.3 would contain compositions (i.e. the Router class is composed of multiple ports). Further supported are inheritance and abstract classes which, in case the example model importer does not recognize them through their spatial information, can either be implied at example model level via annotations or applied to the metamodel directly.

5. The *Model Workbench* supports a semi-automated inference process based on either textual or graphical example models. The metamodel can be accessed and changed by the user and the literature suggests that structural optimizations in the form of inheritance and abstract classes are supported.

The features and capabilities of each tool regarding the metamodel inference process are summarized in table 3.2. Only *metaBup* and the *Model Workbench* are capable of explicitly deriving a metamodel which can be inspected and modified by the user. The other tools build an implicit metamodel which is then used to constrain further sketching activities.

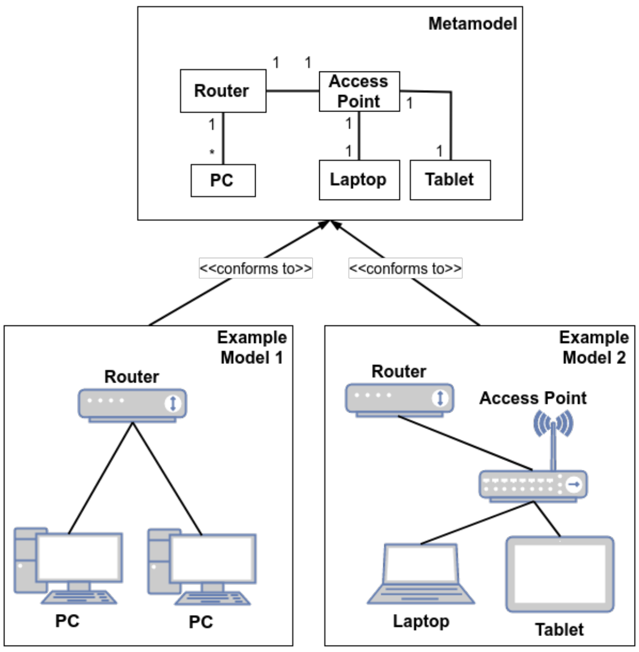
3.3 Co-evolution

To assess the model co-evolution capabilities of every tool, two concrete evolution scenarios are used as follows:

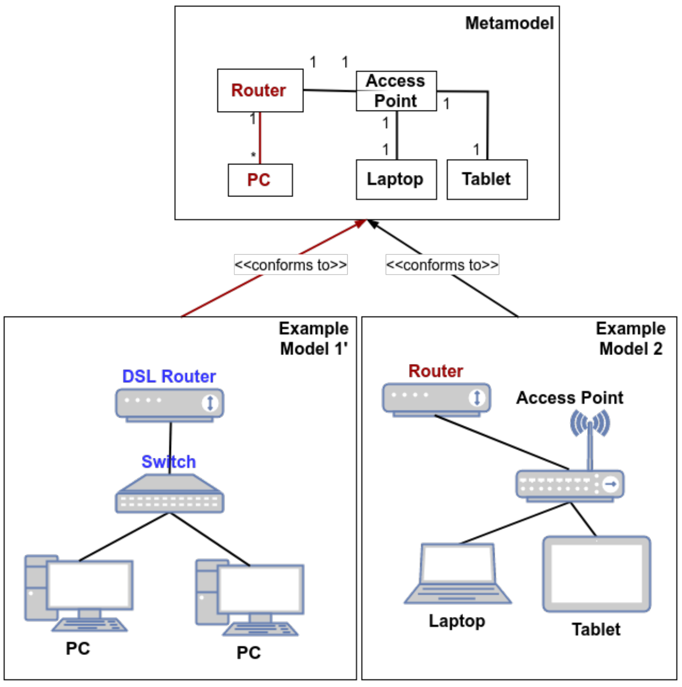
1. **Forward evolution:** A metamodel is inferred from two example models that describe different aspects of the same language. After that, one example model is modified so that the metamodel as well as the other example model need to co-evolve.

2. **Backward evolution:** A metamodel is inferred from a example model. The metamodel is then modified by the user so that the example model needs to co-evolve.

Using the sample language that was already utilized for the models in section 3.1, the test case for forward evolution is given in figure 3.4. Two example models conform to a metamodel. Then, one example model is changed (highlighted in blue) and the conformance relationship breaks, making an evolution of the metamodel necessary. Additionally, since the ”Router” element was renamed, this change also affects the second example model which has to evolve as well. Figure 3.5 depicts the same scenario, however with the change now performed at metamodel level. Again, by renaming the ”Router” element, a breaking and resolvable change was introduced. This requires co-evolving both example models and could possibly performed automatically. Furthermore, a breaking and unresolvable change was done by adding a ”Switch” element between the router and the personal computers. A tool should warn about the connections in the example model that violate this new constraint.

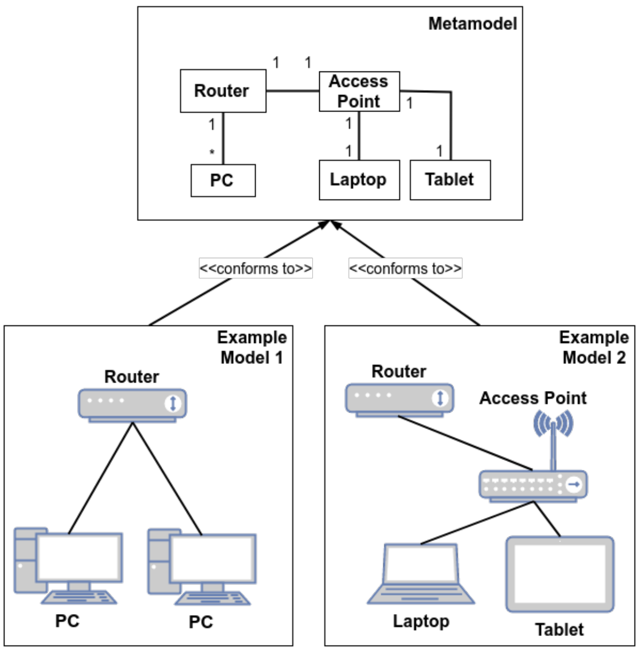


(a) Two example models conforming to a inferred metamodel

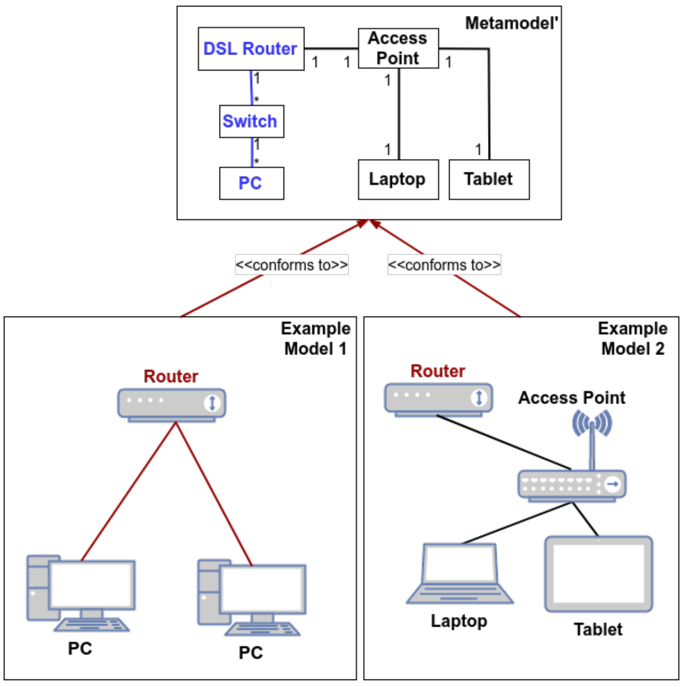


(b) An example model change induces co-evolution

Figure 3.4: Forward evolution example



(a) Two example models conforming to a inferred metamodel



(b) A metamodel change induces co-evolution

Figure 3.5: Backward evolution example

Tool	**Forward Evolution**	**Backward Evolution**	**Classification**
Scribbler	No	Manual	None
MLCBD	Manual	No	None
FlexiSketch	Yes	No	None
metaBup	Yes	Yes	Non-breaking Resolvable Unresolvable
Model Workbench	Yes	Yes	Non-breaking Breaking

Table 3.3: Summary of co-evolution features

The remainder of this section investigates if and how the different tools support the two co-evolution scenarios. Since only *metaBup* and the *Model Workbench* expose the metamodel to the user (see section 3.2), backward evolution is not applicable to *FlexiSketch* and *MLCBD*.

1. *Scribbler* supports manual backward evolution since the metamodel needs to be defined in the Eclipse Modeling Framework by hand and therefore can be changed at any point by the user. However, the mapping between sketched models and metamodel needs to be updated as well. Regarding the scenario depicted in figure 3.5, the user has to manually retain the conformance relationship by renaming the example model elements and introducing the new ”Switch” element.

2. *MLCBD* supports forward evolution by re-iterating the metamodel inference process. In scenario 3.4, after the example model is changed, the metamodel can be updated by regenerating it. However, for the other example model that is affected by the change as well, the user gets a warning about the conflicting types and needs to resolve the issue manually. Backward evolution is not supported due to the metamodel being generated implicitly only.

3. *FlexiSketch* mitigates issues during forward co-evolution by implementing two different synchronization mechanisms. First, for every example model, a corresponding metamodel is stored. Two or more metamodels can me merged into a generalized model as long as only non-breaking changes have to be performed. Second, a lock mechanism can be triggered by the user which disallows any modifications to a metamodel. Therefore, instead of evolving the model, non-conforming parts of an example model will be highlighted. When considering scenario 3.4, *FlexiSketch* would maintain one metamodel for every example model. Changes on one example model will not affect the other one as long as the metamodels have not been merged. If such a merge is executed, two different classes for the ”Router” and the ”DSL Router” element are generated in the metamodel since *FlexiSketch* separates different example models strictly.

4. *metaBup* implements both forward and backward evolution and is capable of handling both scenarios. For 3.4, the rename is treated as a non-breaking issue and applied to the metamodel and all other example models. The new ”Switch” element will be included in the metamodel once it is regenerated. Similar applies for the backward evolution scenario in figure 3.5: renaming an element in the metamodel is classified as non-breaking change and can therefore be applied to every example model. However, when introducing the new ”Switch” element, the tool issues a warning that the first example model does not conform to the metamodel anymore and the user is asked to resolve it manually.

5. The *Model Workbench* supports co-evolution features comparable to *metaBup*. For forward evolution, non-breaking changes like the rename in figure 3.4 are applied automatically upon metamodel regeneration. For every other changes, the user is asked to resolve them manually. Backward evolution also supports non-breaking changes only. In scenario 3.5, the tool notifies the user about the broken conformance relationship between the modified metamodel and the first example model.

Table 3.3 summarizes the co-evolution support of the tools. Again, *metaBup* and the *Model Workbench* feature the most complete implementation and use the commonly used taxonomy of diving the evolution scenarios in non-breaking, breaking and resolvable and breaking and unresolvable at least to some extend. If a change is unresolvable, the user is notified and asked to resolve it manually.

3.4 Tool support

Table 3.4 gives a summary of the tool support analysis which was carried out in multiple dimensions.

	Scribbler	MLCBD	FlexiSketch	metaBup	Model Workbench
Implementation	Java Application	MS Visio plugin	Android App Java Application	EMF plugin	Java EE Application
Integration	EMF	-	-	EMF metaDepth	Self-contained
Usability	Industrial user study	Case study	User study	User study	-
Scalability	Not assessable	Not assessable	Limited	Yes	Not assessable
Collaboration	Client-Server	-	Client-Server	-	Client-Server

Table 3.4: Summary of tool support evaluation

1. **Implementation:** Every tool was implemented with different frameworks and technologies, making the results rather diverse. The area of focus of each tool is reflected in the design and implementation decisions. For example, while *metaBup* emphasizes metamodeling and DSML design aspects and therefore is implemented as a plugin to EMF, a widely used modeling framework, *FlexiSketch* focuses on unconstrained sketching and supports touch-based input devices.

2. **Integration:** Integration with other metamodeling tools is an important factor for the actual value of a tool when used in the scope of real-world projects. It answers the question about how the abstract and concrete syntax can be used after they are defined and considered ready to use. Three possible methods are used in the presented tools: providing an exporter to external metamodeling frameworks, automatically generating such a framework or a self-contained environment where seamless DSML definition and usage is possible. The latter is done by the *Model Workbench* that provides a unified environment to define and use a DSML. *metaBup* is capable to generate a modeling environment with the abstract and concrete syntax as defined by the example models. This is made possible by using EMF's capabilities to build a custom modeling environment out of a metamodel. For *FlexiSketch*, an exporter is planned that generates a MetaEdit+ compatible file for the metamodel. However, no such functionality could be found in the tool. *Scribbler* provides an importer and exporter for EMF: Ecore metamodels can be imported to link elements to shapes and sketched models can be exported to represent them in EMF. *MLCBD* does not feature any kind of integration with other metamodeling tools.

3. **Usability:** The usability of most tools has been evaluated in user studies with varying size and scope. *Scribbler* was made available to industrial users that were asked to use the tool in their daily work and report about their experiences. It was found to be generally helpful and valuable with a good recognition rate of the hand-drawn shapes. *MLCBD* was not evaluated in a user study. However, a case study was presented [15] where a concrete DSML for finite state machines and a network diagram language was developed and the workflow compared with two other metamodeling tools, the Generic Modeling Environment (GME)[2] and the Eclipse Modeling Framework. In summary, it was found that the DSML development complexity has reduced and, in contrast to GME and EMF, no metamodeling expertise is required. Language evolution and metamodel analysis and debug capabilities have been identified as shortcomings of *MLCBD*. The usability of *FlexiSketch* was analyzed in a user study with 17 participants coming from the academic and industrial context. The vast majority of the users found the tool to be useful and stated that they would use a polished version of the tool in their daily work to replace whiteboards and flipcharts. *metaBup* was analyzed in a study with 11 participants with varying background. The participants were asked to design a DSML which requirements were provided as text. The overall usability of the tool was rated as good. Furthermore, no correlation between metamodeling experience and time needed to develop the language was found, suggesting that non-experts benefit greatly from the example-based approach. Finally, the *Model Workbench* was not assessed in any user or case study. Therefore, no statements about its usability can be made.

4. **Scalability:** This criteria is concerned with the adequacy of the tool to handle large real-world data which goes beyond the scope of a demonstration. In the context of Moodling, this means designing a complex language that requires a large amount example models. Due to the unavailability of most tools, this feature was assessed on existing user studies, if available.
The user study conducted with the *Scribbler* tool does not give any information about the size of the used languages and models. Similar holds true for *MLCBD*, where two use cases were presented with a purely demonstrative purpose. The scalability of *FlexiSketch* was not explicitly assessed during the user studies as they focused on the usability of the tool and only feature small-scale examples. Due to the touch-based input method and the limited screen size of touch devices, it can nonetheless be concluded that *FlexiSketch* is mainly suited for quickly sketching high-level models of a language. The user study of *metaBup* included a maximum of six example models to describe the language, with an average of 12 elements and 9 edges per example model. Since the example models get drawn individually in an external general-purpose graphics vector program and with EMF, an established metamodeling framework is used, the scalability of *metaBup* can

[2] `http://www.isis.vanderbilt.edu/Projects/gme`

be considered suitable for industrial-sized projects. Lastly, the *Model Workbench* could not be evaluated with regards to scalability since neither the tool itself or further information is available nor a user study has been conducted.

5. **Collaboration:** Multi-user support is a vital aspect in Moodling since, typically, Moodling involves multiple domain experts and language engineers that actively collaborate together. *Scribbler* supports multiple users working together on one sketch model in parallel. One running instance of the tool functions as server to which the clients connect to. In a multi-user session, mouse movements and events are distributed to all participants, making collaborative sketching possible. *FlexiSketch* has been evolved into *FlexiSketch Team* [94] which extends the original tool with collaboration features similar to *Scribbler*: one tool instance is configured as server to which all clients connect to. To prevent inconsistent states, a locking mechanism prohibits concurrent modification of the same element. Additionally, the sketch sharing mechanism can be disabled by the user to have a private workspace. The *Model workbench* is implemented as a client-server architecture and makes use of modern web technologies. A ReST API connects clients to the server and provides access to the models saved in a database. No further information about collaboration mechanisms could be found that indicates real team support like locking and parallel manipulation of the same elements.

3.5 Comparison

Figure 3.6 shows the results of the evaluation as a radar chart. It points out the maturity of each solution in the individual areas **Unconstrained input, Metamodel generation, Co-evolution** and **Tool support**. For the rating, a point-based system was used, based on the previous analysis. Section 3.1 evaluated the visual expressiveness of the input method. Weak support means that only conceptually simple (that is, no textual annotations or spatial information) example models can be expressed with the provided input method whereas advanced support includes support for visual constructs like spatial relationships. Section 3.3 compared the co-evolution support of the tools. A distinction between forward evolution, backward evolution and an existing evolution classification was made. Weak support in this area means that only one evolution direction is supported while advanced support indicates the presence of all three aforementioned distinct features. Lastly, section 3.4 analyzed the tools with regard to their usability, scalability and collaboration features. Weak support in this area implies that only one of these factors were tested or implemented. In contrast, advanced support means all three areas are covered.

The following findings can be deducted from the comparison:

- In all areas, a high diversity between the tools and their support for specific features exists. This indicates that Moodling is still in its infancy and no best practices have been established yet.
- For sketching example models, multiple methods have been implemented, with either mimicking traditional pen-and-paper sketching on a digital canvas or icon-based editors.
- Only two tools explicitly generate a metamodel that can be inspected and manipulated by the user. The others, albeit allowing to sketch models and implicitly building a metamodel to constrain the sketching process, do not expose it to the user.
- Co-evolution is distinctively diverse, spawning all levels of support. Naturally, a correlation between metamodel generation and evolution maturity exists: if the metamodel can not be manipulated by the user, there is no need for backward evolution.
- Mostly being research prototypes, the tool support itself was the area with the most deficiencies. While all approaches were implemented as either plugins or standalone applications, most of them are unavailable, making a throughout evaluation difficult.

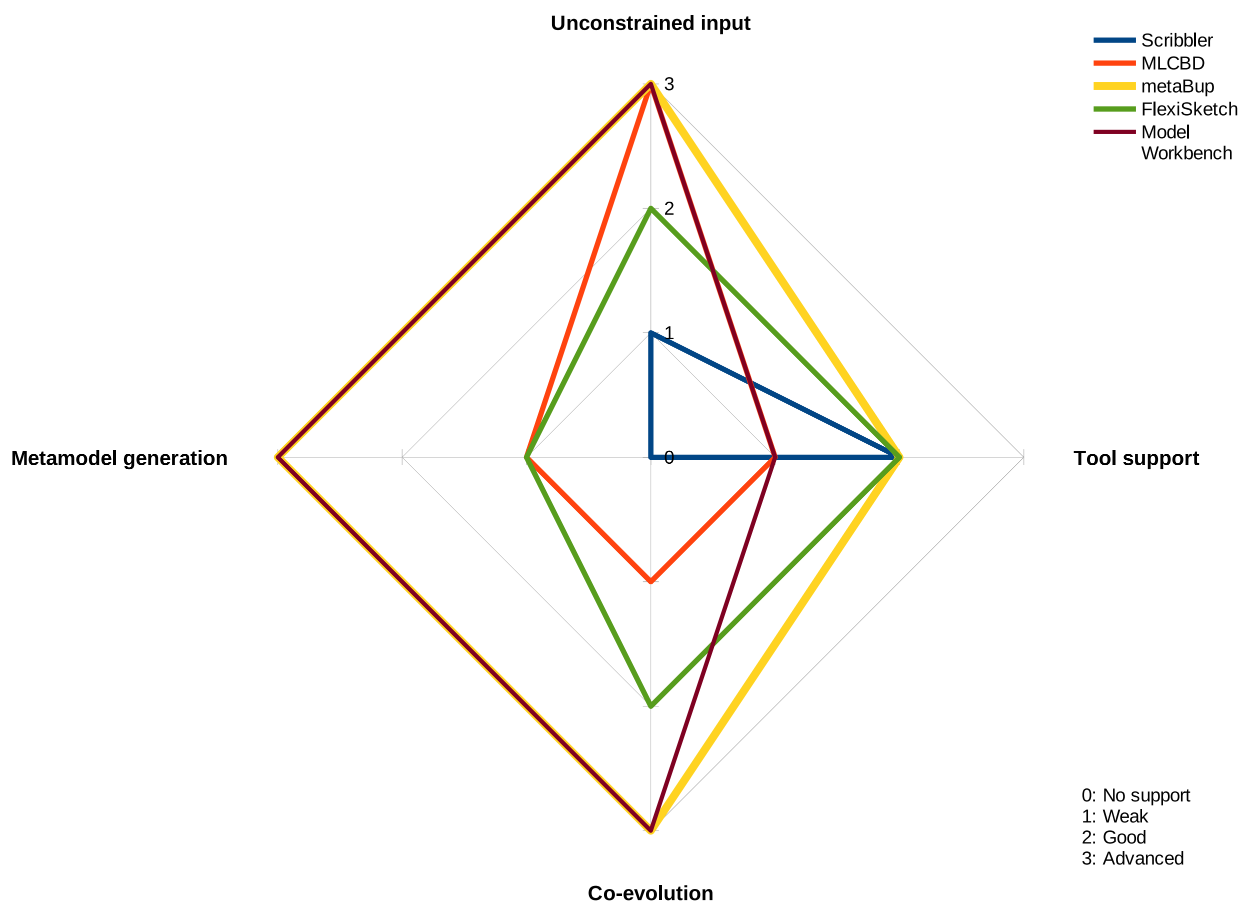


Figure 3.6: Summary of strengths and weaknesses of solutions

4
Conclusion

This paper has given an overview of the approach to create a DSML metamodel definition in a bottom-up way, starting from a set of example models. Furthermore, it investigated existing solutions by classifying them using several criteria: sketch recognition, metamodel inference, evolution of the DSML, environment integration and tool support. These criteria were used to analyze and compare the different solutions in an effort to find open research questions. We conclude the following:

– Full automation for both metamodel inference and co-evolution is not possible due to the fact that the user's intention has to be captured and manual intervention is required for ambiguous scenarios. Further work in this area is not considered conducive.
– Most solutions force the language designer to specify a single shape for each metamodel concept. Therefore, the mapping between concrete and abstract syntax is manually defined as a one-to-one relationship. Once introduced, a concrete syntax representation for a concept is difficult to alter.
– Support for model co-evolution exists within the scope of metamodel refactorings, but is mostly basic without structured approaches implemented. No clear distinction between forward and backward evolution is made. Missing support for automated co-evolution limits the user's ability to freely manipulate both example models and the metamodel to quickly adapt it to changing DSML requirements. Furthermore,
– No solution provides seamless integration of language design and usage. Mostly, a strict separation between the metamodel construction and the model instantiation environment exists. This makes it difficult to use the generated metamodel in the early design phases to acquire knowledge about its usability and adequacy. Also, some solutions rely on exporting the metamodel to external metamodeling tools and therefore discourage the user to continue evolution within the Moodling framework once such an export has been performed. In order to implement a holistic agile DSML design process, co-evolution must consider the complete development chain, from the example models to the metamodeling environment.
– Tool support was perceived as rather poor since the majority of them are either prototypes or unavailable. Adoption of the Moodling approach would benefit from easy to use, mature and stable tools.
– The usability and added value of bottom-up metamodel construction was only investigated in small-scale user experiments. Currently, no detailed experience reports in the scope of large projects exist, suggesting that Moodling has not attracted attention in the industrial environment yet.

Further research concerning these issues will help remedy the remaining problems that inhibit wide-spread acceptance of Moodling as a method for developing domain-specific modeling languages.

Bibliography

[1] Scott W Ambler. Agile model driven development is good enough. *IEEE Software*, 20(5):71–73, 2003.

[2] Scott W Ambler. *The object primer: Agile model-driven development with UML 2.0.* Cambridge University Press, 2004.

[3] Islem Baki and Houari Sahraoui. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):20, 2016.

[4] Zoltán Balogh and Dániel Varró. Model transformation by example using inductive logic programming. *Software & Systems Modeling*, 8(3):347–364, 2009.

[5] Christian Bartelt, Martin Vogel, and Tim Warnecke. Collaborative creativity: From hand drawn sketches to formal domain specific models and back again. In A. Notle, M. Prilla, P. Rittgen, and S. Oppl, editors, *Proceedings of the International Workshop on Models and their Role in Collaboration at the ECSCW 2013 (MoRoCo 2013)*, pages 25–32, 09 2013.

[6] Christian Bartelt, Martin Vogel, and Tim Warnecke. Scribbler: From collaborative sketching to formal domain specific models and back again. In *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems (Models 2013)*, 10 2013.

[7] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. Mdeforge: an extensible web-based modeling platform. In *CloudMDE@ MoDELS*, pages 66–75, 2014.

[8] Michel Beaudouin-Lafon and Wendy Mackay. Prototyping tools and techniques. In Julie A. Jacko and Andrew Sears, editors, *The Human-computer Interaction Handbook*, pages 1006–1031. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 2003.

[9] Aude Billard, Sylvain Calinon, Rüdiger Dillmann, and Stefan Schaal. *Robot Programming by Demonstration*, pages 1371–1394. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] F. Brooks and H. J. Kugler. *No silver bullet.* April, 1987.

[11] Qi Chen, John Grundy, and John Hosking. An e-whiteboard application to support early design-stage sketching of UML diagrams. In *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*, pages 219–226. IEEE, 2003.

[12] Qi Chen, John Grundy, and John Hosking. Sumlow: Early design-stage sketching of UML diagrams on an e-whiteboard. *Softw. Pract. Exper.*, 38(9):961–994, July 2008.

[13] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: How and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 557–566, New York, NY, USA, 2007. ACM.

[14] Hyun Cho. Creating domain-specific modeling languages using by-demonstration technique. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 211–212, New York, NY, USA, 2011. ACM.

[15] Hyun Cho. *A Demonstration-based Approach for Domain-specific Modeling Language Creation.* PhD thesis, University of Alabama, Tuscaloosa, AL, USA, 2013. AAI3562407.

[16] Hyun Cho and Jeff Gray. Design patterns for metamodels. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 25–32, New York, NY, USA, 2011. ACM.

[17] Hyun Cho, Jeff Gray, and Eugene Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, MiSE '12, pages 22–28, Piscataway, NJ, USA, 2012. IEEE Press.

[18] Hyun Cho, Yu Sun, Jeff Gray, and Jules White. Key challenges for modeling language creation by demonstration. In *ICSE 2011 Workshop on Flexible Modeling Tools, Honolulu HI*, 2011.

[19] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[20] Paul Corey and Tracy Hammond. Gladder: Combining gesture and geometric sketch recognition. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1788–1789. AAAI Press, 2008.

[21] Jonathan Corley, Eugene Syriani, Huseyin Ergin, and Simon Van Mierlo. Cloud-based multi-view modeling environments. In *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, pages 120–139. IGI Global, 2016.

[22] Adrien Coyette, Sascha Schimke, Jean Vanderdonckt, and Claus Vielhauer. Trainable sketch recognizer for graphical user interface design. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-computer Interaction*, INTERACT'07, pages 124–135, Berlin, Heidelberg, 2007. Springer-Verlag.

[23] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[24] James Davis. Gme: The generic modeling environment. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 82–83, New York, NY, USA, 2003. ACM.

[25] R. Davis. Magic paper: Sketch-understanding research. *Computer*, 40(9):34–41, Sept 2007.

[26] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[27] J. Denil, R. Salay, C. Paredis, and H. Vangheluwe. Towards agile model-based systems engineering. *Flexible Model Driven Engineering Proceedings (FlexMDE 2017)*, 2017. to appear.

[28] Guihuan Feng, Christian Viard-Gaudin, and Zhengxing Sun. On-line hand-drawn electric circuit diagram recognition using 2D dynamic programming. *Pattern Recognition*, 42(12):3215–3223, 2009.

[29] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[30] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering*, 2017.

[31] Adnane Ghannem. *Example-based model refactoring using heuristic search*. PhD thesis, École de technologie supérieure, 2015.

[32] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. Model refactoring using examples: a search-based approach. *Journal of Software: Evolution and Process*, 26(7):692–713, 2014.

[33] Vinod Goel. *Sketches of Thought: A Study of the Role of Sketching in Design Problem-solving and Its Implications for the Computational Theory of the Mind*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 1991. UMI Order No. GAX92-28664.

[34] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Using free modeling as an agile method for developing domain specific modeling languages. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 24–34. ACM, 2016.

[35] Cláudio Gomes, Joachim Denil, and Hans Vangheluwe. Causal-block diagrams. Technical report, University of Antwerp, 2016.

[36] Paola Gómez, Mario E Sánchez, Hector Florez, and Jorge Villalobos. An approach to the co-creation of models and metamodels in enterprise architecture projects. *Journal of Object Technology*, 13(3):2–1, 2014.

[37] Daniel Conrad Halbert. *Programming by Example*. PhD thesis, University of California, Berkeley, 1984. AAI8512843.

[38] Tracy Hammond and Randall Davis. Tahuti: A geometrical sketch recognition system for UML class diagrams. In *ACM SIGGRAPH 2006 Courses*, page 25. ACM, 2006.

[39] Tracy Hammond, Brandon Paulson, and Brian Eoff. Eurographics tutorial on sketch recognition. In *Eurographics (Tutorials)*, pages 1–4, 2009.

[40] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[41] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, October 2004.

[42] Regina Hebig, Holger Giese, Florian Stallmann, and Andreas Seibel. On the complex nature of MDE evolution. In *International Conference on Model Driven Engineering Languages and Systems*, pages 436–453. Springer, 2013.

[43] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. Approaches to Co-Evolution of Metamodels and Models: A Survey. *IEEE Transactions on Software Engineering*, 43(5):396–414, 2017.

[44] Nicolas Hili. A metamodeling framework for promoting flexibility and creativity over strict model conformance. In *Flexible Model Driven Engineering Workshop*, volume 1694, pages 2–11. CEUR-WS, 2016.

[45] Shuhei Hiya, Kenji Hisazumi, Akira Fukuda, and Tsuneo Nakanishi. clooca: Web based tool for domain specific modeling. In *Demos/Posters/StudentResearch@ MoDELS*, pages 31–35, 2013.

[46] Javier Luis Cánovas Izquierdo, Jordi Cabot, Jesús J López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan De Lara. Engaging end-users in the collaborative development of domain-specific modelling languages. In *International Conference on Cooperative Design, Visualization and Engineering*, pages 101–110. Springer, 2013.

[47] Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R. Bryant. Mars: A metamodel recovery system using grammar inference. *Inf. Softw. Technol.*, 50(9-10):948–968, August 2008.

[48] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Model transformation by-example: a survey of the first wave. In *Conceptual Modelling and Its Theoretical Foundations*, pages 197–215. Springer, 2012.

[49] Steven Kelly and Juha-pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. 04 2008.

[50] Dimitrios S Kolovos, Nicholas Drivalos Matragkas, Horacio Hoyos Rodríguez, and Richard F Paige. Programmatic muddle management. *XM@ MoDELS*, 1089:2–10, 2013.

[51] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, Dec 2006.

[52] James A. Landay. Silk: Sketching interfaces like krazy. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 398–399, New York, NY, USA, 1996. ACM.

[53] Craig Larman and Victor R Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[54] Tessa Ann Lau. *Programming by Demonstration: A Machine Learning Approach*. PhD thesis, University of Washington, 2001. AAI3013992.

[55] Henry Lieberman. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[56] Jesús J. López-Fernández. *An agile process for the example-driven development of modelling languages and environments*. PhD thesis, Autonomous University of Madrid, May 2017.

[57] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. Example-driven meta-model development. *Softw. Syst. Model.*, 14(4):1323–1347, October 2015.

[58] Jesús J López-Fernández, Antonio Garmendia, Esther Guerra, and Juan de Lara. Example-based generation of graphical modelling environments. In *European Conference on Modelling Foundations and Applications*, pages 101–117. Springer, 2016.

[59] Jesús J. López-Fernández, Antonio Garmendia, Esther Guerra, and Juan de Lara. An example is worth a thousand words: Creating graphical modelling environments by example. *Software & Systems Modeling*, Nov 2017.

[60] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. Example-based validation of domain-specific visual languages. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 101–112, New York, NY, USA, 2015. ACM.

[61] Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *4 th Workshop on Domain-Specific Modeling*, 2004.

[62] Raphael Mannadiar and Hans Vangheluwe. Domain-specific engineering of domain-specific languages. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, page 11. ACM, 2010.

[63] Raphael Mannadiar and Hans Vangheluwe. Debugging in domain-specific modelling. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 276–285, Berlin, Heidelberg, 2011. Springer-Verlag.

[64] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60, 2014.

[65] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[66] Reza Matinnejad. Agile model driven development: An intelligent compromise. In *Software Engineering Research, Management and Applications (SERA), 2011 9th International Conference on*, pages 197–202. IEEE, 2011.

[67] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[68] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[69] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Sci. Comput. Program.*, 76(12):1223–1246, December 2011.

[70] Mark Minas. Generating meta-model-based freehand editors. *Electronic Communications of the EASST*, 1, 2007.

[71] Parastoo Mohagheghi, Miguel A. Fernandez, Juan A. Martell, Mathias Fritzsche, and Wasif Gilani. MDE adoption in industry: Challenges and success criteria. In Michel R. Chaudron, editor, *Models in Software Engineering*, pages 54–59. Springer-Verlag, Berlin, Heidelberg, 2009.

[72] Daniel Moody. The physics of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

[73] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[74] Tom Y. Ouyang and Randall Davis. Chemink: A natural real-time recognition system for chemical drawings. In *Proceedings of the 16th International Conference on Intelligent User Interfaces*, IUI '11, pages 267–276, New York, NY, USA, 2011. ACM.

[75] Frauke Paetsch, Armin Eberlein, and Frank Maurer. Requirements engineering and agile software development. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, pages 308–313. IEEE, 2003.

[76] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995.

[77] Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Inc., 2010.

[78] Bastian Roth. *Beispielgetriebene Entwicklung domnenspezifischer Modellierungssprachen*. PhD thesis, University of Bayreuth, 2014.

[79] Bastian Roth, Matthias Jahn, and Stefan Jablonski. A method for directly deriving a concise meta model from example models, 2013.

[80] Bastian Roth, Matthias Jahn, and Stefan Jablonski. On the way of bottom-up designing textual domain-specific modelling languages. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, DSM '13, pages 51–56, New York, NY, USA, 2013. ACM.

[81] Bastian Roth, Matthias Jahn, and Stefan Jablonski. Rapid design of meta models. *International Journal on Advances in Software*, 7:31 – 43, 2014.

[82] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[83] Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. Bottom-up meta-modelling: An interactive approach. In *International Conference on Model Driven Engineering Languages and Systems*, pages 3–19. Springer, 2012.

[84] Jean-Sébastien Sottet and Nicolas Biri. JSMF: a javascript flexible modelling framework. In *FlexMDE@MoDELS*, volume 1694 of CEUR Workshop Proceedings, pages 42–51, 2016.

[85] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[86] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment. In *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, pages 21–25, 2013.

[87] Yentl Van Tendeloo, Simon Van Mierlo, Bart Meyers, and Hans Vangheluwe. Concrete syntax: A multi-paradigm modelling approach. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 182–193, New York, NY, USA, 2017. ACM.

[88] Dániel Varró. Model transformation by example. In *International Conference on Model Driven Engineering Languages and Systems*, pages 410–424. Springer, 2006.

[89] Martin Vogel, Tim Warnecke, Christian Bartelt, and Andreas Rausch. Scribbler—drawing models in a creative and collaborative environment: from hand-drawn sketches to domain specific models and vice versa. In *Proceedings of the Fifteenth Australasian User Interface Conference-Volume 150*, pages 93–94. Australian Computer Society, Inc., 2014.

[90] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*, pages 1–17, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[91] Yin Yin Wong. Rough and ready prototypes: Lessons from graphic design. In *Posters and Short Talks of the 1992 SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 83–84, New York, NY, USA, 1992. ACM.

[92] Dustin Wüest, Norbert Seyff, and Martin Glinz. Flexisketch: A mobile sketching tool for software modeling. In David Uhler, Khanjan Mehta, and Jennifer L. Wong, editors, *Mobile Computing, Applications, and Services: 4th International Conference, MobiCASE 2012, Seattle, WA, USA, October 11-12, 2012. Revised Selected Papers*, pages 225–244, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[93] Dustin Wüest, Norbert Seyff, and Martin Glinz. Semi-automatic generation of metamodels from model sketches. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 664–669, Nov 2013.

[94] Dustin Wüest, Norbert Seyff, and Martin Glinz. Sketching and notation creation with FlexiSketch Team: Evaluating a new means for collaborative requirements elicitation. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pages 186–195, Aug 2015.

[95] Dustin Wüest, Norbert Seyff, and Martin Glinz. Flexisketch: a lightweight sketching and metamodeling approach for end-users. *Software & Systems Modeling*, Sep 2017.

[96] Y. Zhang and S. Patel. Agile model-driven development in practice. *IEEE Software*, 28(2):84–91, March 2011.