UNIVERSITEIT ANTWERPEN
Faculty of Sciences
Department of Mathematics and Computer Science
June 2018

Moodling: An Integrated Approach Towards Example-based Domain-Specific Language Design with Focus on Agility

---

Author:
Lucas Heer

Promoter:
Prof. Dr. Hans Vangheluwe
Advisers:
Dr. Simon Van Mierlo
Yentl Van Tendeloo

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science: Computer Science
in the
Modelling, Simulation and Design Lab
Department of Mathematics and Computer Science

Abstract

Domain-Specific Modeling Languages (DSMLs) are increasingly used by system engineers to specify and document both the structure and the 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 an increased level of abstraction. However, the development of a DSML is a complex task and often requires experienced language engineers. To tackle this issue, methods have been proposed that automatically derive a DSML definition from example models. This allows engineers to express the requirements of the language in an intuitive way and minimized the need for in-depth language engineering knowledge. Currently, such example models are either sketched in dedicated drawing applications and imported in metamodeling environments, or drawing applications are extended with metamodeling features. Both approaches reduce the agility of the language design process by introducing a gap between the example model design and instance modeling phases. As a result, the co-evolution of example models, the metamodel and instance models is inhibited. This thesis proposes a solution for example-driven DSML design where sketching and metamodeling activities are integrated in a single process. Example models are elevated to the primary description of the language. It is more flexible, reduces the complexity of the co-evolution issue and lowers the cognitive load for the user. The approach is evaluated with a prototypical implementation in the Modelverse, a multi-paradigm metamodeling environment.

Acknowledgements

First of all, I would like to thank my promoter, Hans Vangheluwe. During my master studies, he introduced me to the field of system's modeling and simulation and provided valuable input during my research internship and my Master's thesis. Also, I would like to thank my supervisors Dr. Simon Van Mierlo and Yentl Van Tendeloo for answering my questions, helping me whenever needed and reviewing my work. Lastly, I would like to thank my family and friends who have been a great support during my whole study career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the major challenges in modern systems development is to master the growing complexity of software-intensive systems, where hardware components are tightly integrated with a software controller. 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 [12]. This chapter describes fundamentals that act as basis for the thesis and formulates research goals. First, section 1.1 introduces Model-Driven Engineering as an approach to master the accidental complexity of software development. Then, 1.2 gives an overview of example-driven modeling language design, where a domain-specific modeling language is designed based on example models. Lastly, 1.3 presents the research agenda of this thesis.

1.1 Model-Driven Engineering

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 [55]. Beside general-purpose modeling languages such as Petri Nets [89] or Statecharts [44] 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 [59]. 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 [84]. 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) [45]. 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 [75], in which case it is called denotational semantics or by explicitly giving the execution rules. In this case, it is called operational semantics [74].

1.1.1 Model Transformations

Model transformations are used to define the relationship between models and the languages the models conform to. They have been referred to as "the Heart and Soul of Model-Driven Software Development" [104] and are considered "undoubtedly the most profound aspect of model-driven software development" [81]. Model transformations take one or more input models and generate one or more output models, according to the definition of the transformation. Such a definition consists out of a set of *transformation rules* that describe how constructs of the source language are transformed to constructs of the target language. For defining such transformation rules, multiple ways exist. For example, they can be specified manually using an imperative programming language. Here, the code needs access to the model to query and manipulate it according to the transformation [110]. Another possibility to define transformation rules make use of graph grammars [57] and declarative pattern languages [108].

Model transformations can be classified into *endogenous* and *exogenous* transformations, sometimes also referred to as model *rephrasing* and *translation*, respectively [81]. Examples of endogenous transformations include optimizations, model refactorings and normalization. Examples for exogenous transformations are synthesis, migration and reverse engineering. Furthermore, model transformation can change models *in-place*, in which case source and target models are the same, and *out-place*, where new models are created.

1.1.2 The FTG+PM Formalism

The *FTG+PM* is a framework to guide developers through the MDE lifecycle, including activities such as requirements development, DSML design, simulation, analysis and code generation [71]. It consists of the *Formalism Transformation Graph* (FTG) and the *Process Model* (PM). While the FTG describes the different formalisms and their relationships by means of transformations, the PM models the control and data flow between the different MDE activities. In essence, the FTG is a hypergraph with languages as nodes and transformations as edges. The PM is a subset of the UML 2.0 activity diagram. Hereby, all activities in the PM are typed by a transformation in the FTG and all model artifacts in the PM conform to a language in the FTG.
The FTG+PM has been implemented in *AToMPM, A Tool for Multi-Paradigm Modeling* [109], which supports executing the defined transformations by using its explicitly modeled rule-based graph transformation language [72]. Furthermore, it has been successfully applied to a non-trivial use case from the automotive domain [90]. We use FTG+PM models throughout this thesis to describe workflows and processes.

1.1.3 Top-Down DSML Design

Within the MDE community, the most commonly used process to develop a DSML from scratch is executed in a top-down manner [73][107]. It typically starts with gathering and analyzing requirements from *domain experts*. These 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 commonly involved in the top-down DSML creation process. This process is modeled as an FTG+PM and shows the different activities and artifacts involved in the described process. Furthermore, we have annotated each activity to signalize if it is executed by a domain expert or a language engineer.

1.2 Example-driven Metamodel Development

The described DSML design process exhibits flaws that can be attributed to the centralization of the metamodel construction and to its strict top-down nature. Most noticeably, 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

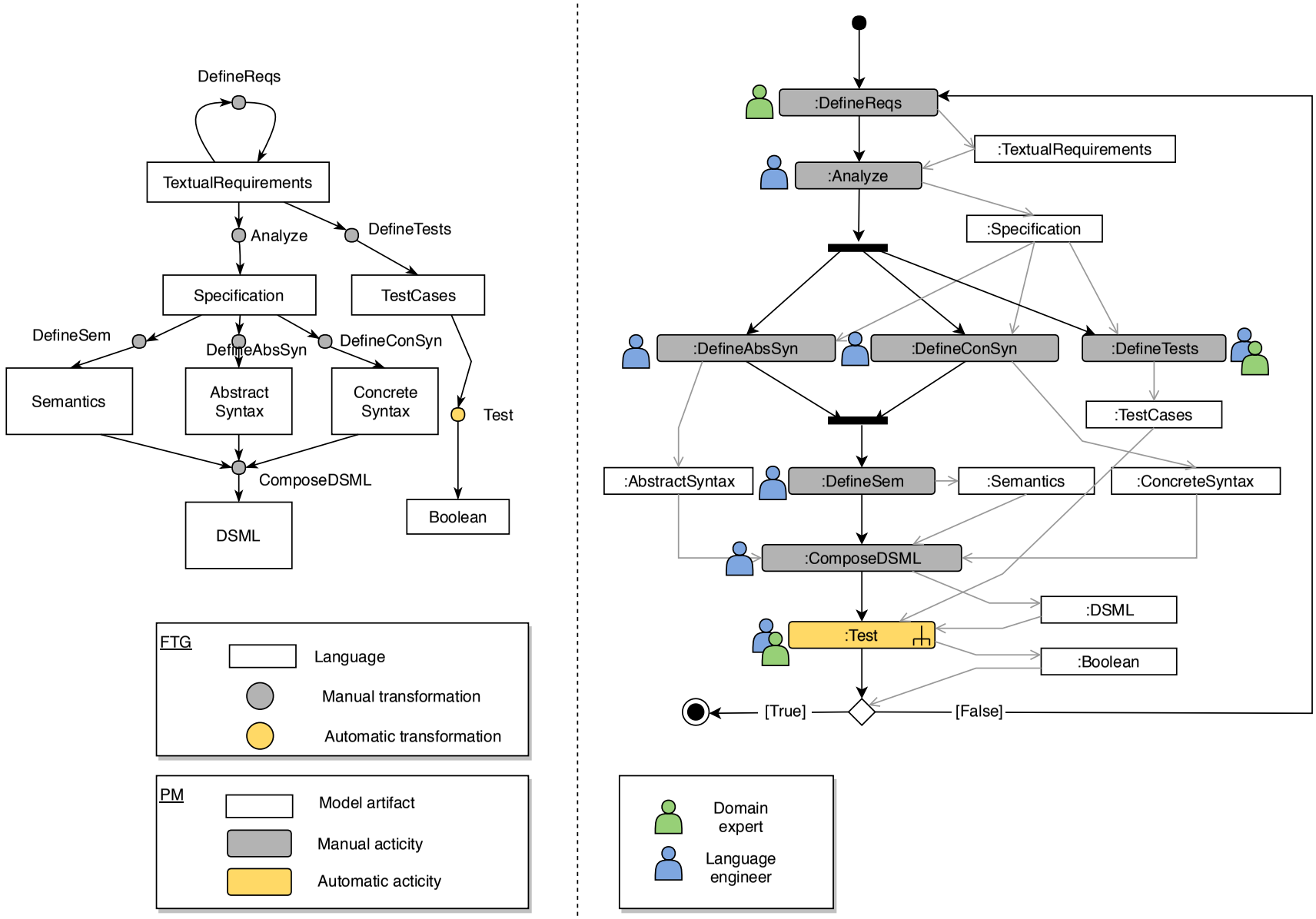
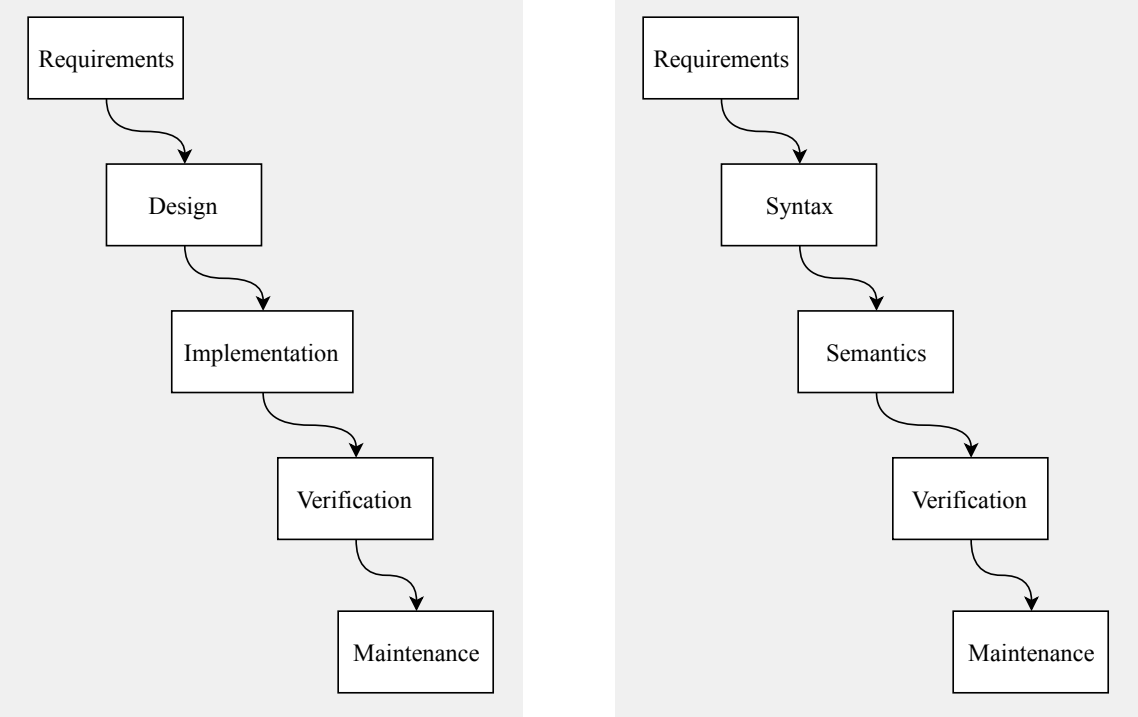


Figure 1.1: FTG+PM depicting the traditional top-down approach for DSML design

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 [67]. 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 [35][120][15]. Beaudouin-Lafon et al. describe pen and paper prototyping as the fastest, cheapest and most widespread prototyping method for interactive system design [10].

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 [51]. This results in an expertise gap between the domain experts and the language engineers that introduces communication problems and might be difficult to overcome [20]. It also shifts a lot of the work to the language engineer and creates an imbalance in the workload of the involved engineers [66]. 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 [118][84].



(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

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 [102]. 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. 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 [96].
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 [63]. 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 [77]. 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 [94]. 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* and has first been described by Ambler [2][1]. Since then, several concrete processes have been proposed [78] as well as a multitude of applications [28], with preliminary research and industrial case studies showing promising results [1][126]. 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.

A promising approach 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. Such a process bears similarities with the idea of *Programming by Demonstration* (PbD, sometimes also called *Programming By Example*, PbE), a technique that appeared in software development research in the mid 1980s [41][65]. Hereby, 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. 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 [24]. Several uses for PbD have been proposed, most noticeably in the field of robotics [11] and machine learning [64].

By deriving a modeling language from examples, the principal idea of PbD has been transferred to the field of DSML design. Figure 1.3 gives a generic FTG+PM for a DSML design process based on example models. Since the exact process differs between various existing approaches, we only captured a high-level overview of the process and refer to chapter 2 for an extensive analysis of how this process is implemented in different tools. Compared to the bottom-up DSML design process shown in 1.1, no formal requirements engineering phase is preceding all other activities. Instead, engineers express requirements directly by creating a set of example models. These example models give an initial idea of how models of the language could look like and are the equivalent of informal sketches that are typically created during brainstorming and initial system design phases. These example models are then used to automatically generate the abstract syntax of the DSML, since they give information about the structure and the visual representation of the language. Depending on the approach, this metamodel can be exported to a format that can be imported into a metamodeling environment to obtain a modeling tool which is tailored to the DSML. Another possibility is a self-contained environment where example models and instance models are created within the same tool and the abstract syntax is directly used to constrain the modeling activity. In this case, the metamodel does not have to be exported. The example model artifacts are input to the modeling phase since the concrete syntax of the language are implicitly defined by them. Therefore, the modeling environment requires access to the example models to render models on screen. During modeling, the adequacy of the generated DSML is verified. This step is similar to the testing activity in 1.1 with the difference that no formal requirements were defined in the first place. As a result, this verification step can be as simple as the engineers checking if all required constructs can be modeled. If this is not the case, the language must be revised by either adapting the example models or the metamodel. If the example models are changed, the metamodel typically has to

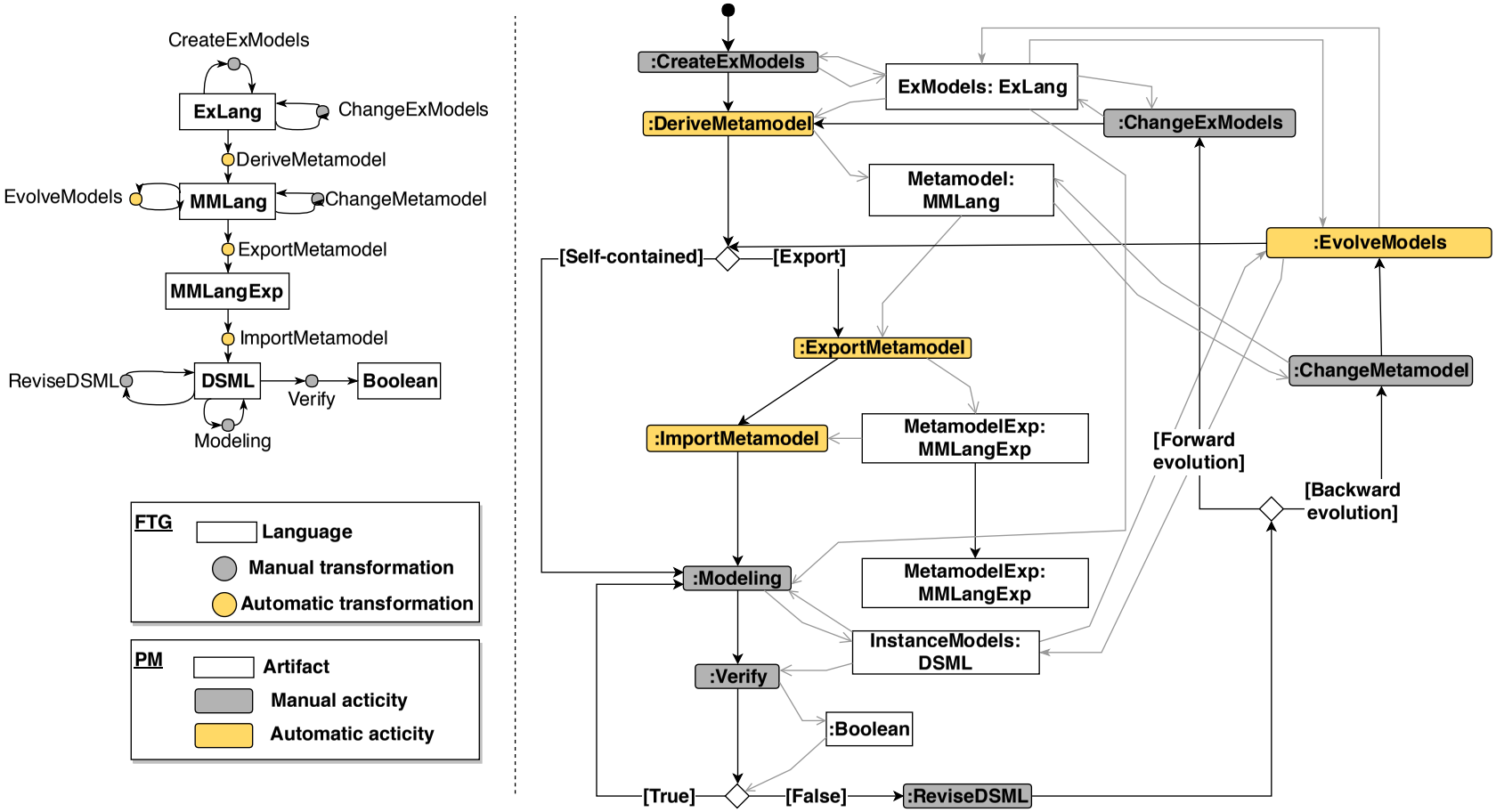


Figure 1.3: FTG+PM for the bottom-up, example-driven approach to DSML design

be re-generated. In case the metamodel is changed, both the instance- and example models have to co-evolve. Note that we assume an automated co-evolution of instance models, although this depends on the implementation and the changes performed to the metamodel, as some of them might require manual user intervention. This example-driven design process 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 DSML elements creation, they can supervise the whole process, define domain concepts, further refine the metamodel and participate in reviewing the quality of the DSML during the validation and revision process. Therefore, DSML development becomes more approachable for non-experts that would be otherwise deterred by the formality and complexity of the top-down language design approach.

Various names have been proposed and used interchangeably 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" and "bottom-up designing domain-specific modeling languages". For the remainder of this thesis, we will generally adhere to "example-driven DSML design" to describe all approaches where a DSML is designed by using examples.

1.3 Research Agenda

We believe that the example-driven DSML design process as described in 1.2 does not fully remedy all issues of the top-down DSML design approach and still resembles the waterfall model as shown in figure 1.2b. In particular, the process is limited in its agility, has a long iteration loop for integrating new requirements and requires metamodeling expertise due to the fact that a metamodel is explicitly generated. In chapter 2, we systematically analyze and compare existing solutions and detail on such limitations. The research goal of this thesis is to investigate how current limitations in existing example-driven DSML design approaches can be overcome. Our focus lies on improving the agility and reducing the cognitive load of the process by integrating it in a single metamodeling environment. Furthermore, we will investigate how the problem of language evolution can be solved by inferring language constraints on-the-fly rather than generating an explicit metamodel. Therefore, we can formulate the central research goal as:

> *Current example-driven DSML design processes focus on the automated generation of a metamodel as primary description for language structure and make a strict separation between the language design and usage phases, often implemented in different environments. This increases the cognitive load for the user, limits the agility and makes language evolution driven only by*

manipulating the example models difficult. Is it possible to design an agile process that is centric to the example models and implement it in a single environment?

The remainder of this thesis is structured as follows. Chapter 2 gives an overview of the already conducted research in the field of example-driven DSML design. It presents existing solutions and analyzes their advantages and disadvantages. This analysis will later help to compare our proposed solution to the existing ones. Chapter 3 presents the theoretical foundations and the design of our approach. Chapter 4 gives details about the prototypical implementation of the previously presented approach in the Modelverse, a multi-paradigm metamodeling environment. Chapter 5 analyzes our solution with respect to existing tools and shows the novel advancements and points out limitations. Lastly, chapter 6 concludes the thesis and suggests future work.

Chapter 2

Related Work

This chapter presents a literature review of existing approaches and tools that propose an example-driven DSML creation process. It is a shortened reproduction of our previous work [48] and is included in this thesis for the sake of completeness. Compared to the original material, two changes were done to fit in the context of this thesis: chapter 2 was removed, since it only gives extensive background information about individual challenges in example-driven DSML design. Relevant parts that improve comprehension and explain key aspects were inserted into the analysis chapter where needed. Furthermore, the challenge of "Metamodel Generation" was renamed to "Metamodeling support", since not all approaches focus on the generation of a language metamodel. Lastly, the test cases for co-evolution support were extended by breaking and unresolvable scenarios to allow for a more fine-grained analysis and comparison.

2.1 Overview of Existing Tools

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 [81]. As a consequence, research has been conducted that investigates methods to determine such transformation rules by example [116][3][4][53]. Other areas include model refactorings [34][33] and DSML validation [70]. Furthermore, the tool *Muddles* [56] 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 [49][105][83], others let the metamodel evolve together with the instance models [36][38]. Lastly, it has been tried to make MDE more suitable for agile development processes by implementing team collaboration and multi-view features [23][32]. For example, *AToMPM* is a multi-paradigm modeling tool which elevates web-based technologies for a multi-user cloud-based architecture [109]. Other tools which provide development environments running in web browsers are *Clooca* [50], *WebGME* [76] and *MDEForge* [9]. 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 2.1. *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 of a predefined 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, 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.

[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`

[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`

Name	Year	Implemented	Resources
MLCBD	2012-2013	Yes (MS Visio plugin)	[17][19][16]
Scribbler	2013	Yes (Standalone)	[7][8][117]
FlexiSketch	2012-2015	Yes (Android App)	[121][122][123][124]
metaBup	2012-2017	Yes (Eclipse plugin)	[67][69][66][51][103][68]
Model Workbench	2013-2014	Yes (Standalone)	[99][98][101][100]

Table 2.1: Overview of investigated tools

2.2 Analysis

In this chapter, the quality of each solution will be analyzed with respect to individual areas as identified in our preliminary work on example-driven DSML design [48], namely **Unconstrained input**, **Metamodeling support**, **Co-evolution** and **Tool support**. 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.

2.2.1 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 [86]. Although rectangular shapes are the most predominantly used notation in visual languages [95], it can not be anticipated what constructs the user will use for the example models. Therefore, a good input method provides a wide variety of visual notations and is able to recognize and differentiate between them.

In the existing tools, input methods can be distinguished between approaches that mimic pen-and-paper drawing and approaches that use general-purpose sketching tools like *Microsoft Visio*

or *yED*[6]. For free-hand drawing, the drawn shapes must be recognized and classified as concrete syntax representations of metamodel elements. 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 [93] to electric circuit design [30]. In the scope of systems modeling, it has been used for example to recognize hand-drawn UML shapes [42][13] or user interface design [61]. 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 [43]. 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 [22]. 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 example-driven DSML design where the goal is to give the user as much freedom as possible while sketching the example models of the DSML.

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 2.1. 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 2.2, 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 2.3 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 using a human input device such as a mouse or a digital drawing tablet. The 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*

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

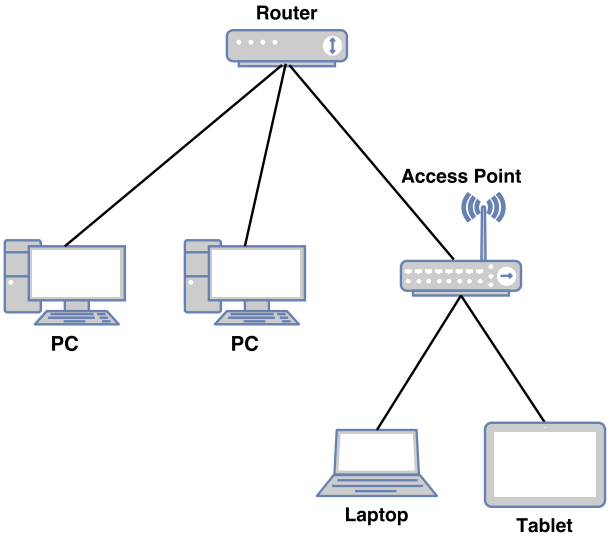


Figure 2.1: The most basic example model with entities and relations

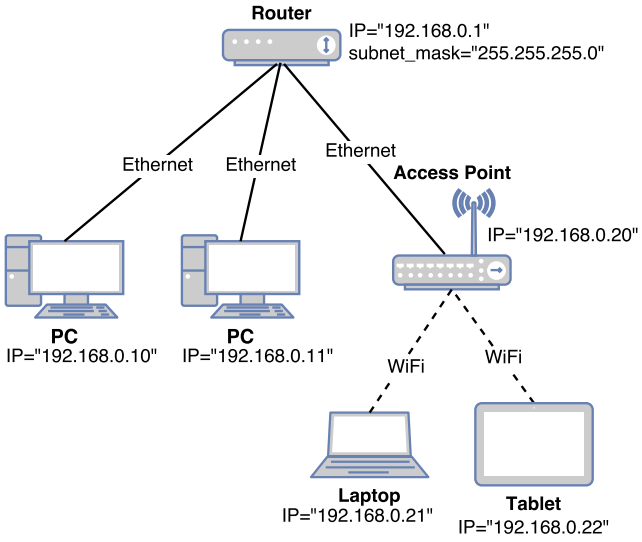


Figure 2.2: More advanced example model with annotations

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 2.1. However, no evidence could be found that implies support for more advanced constructs like annotations or spatial relationships as shown in the figures 2.2 and 2.3, respectively.

2. *MLCBD* is capable of both sketch recognition of hand-drawn shapes and modeling with predefined icons. It comes with an authoring tool to register new shapes that should be recognized later. The sketch recognition itself is based on *SUMLOW*, a tool for recognizing UML diagrams sketched on an electronic whiteboard [14]. *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 predefined shapes that cover commonly used shapes such as circles, rectangles and arrows. Analogous to *SUMLOW*, the user has to remember the order of the strokes that

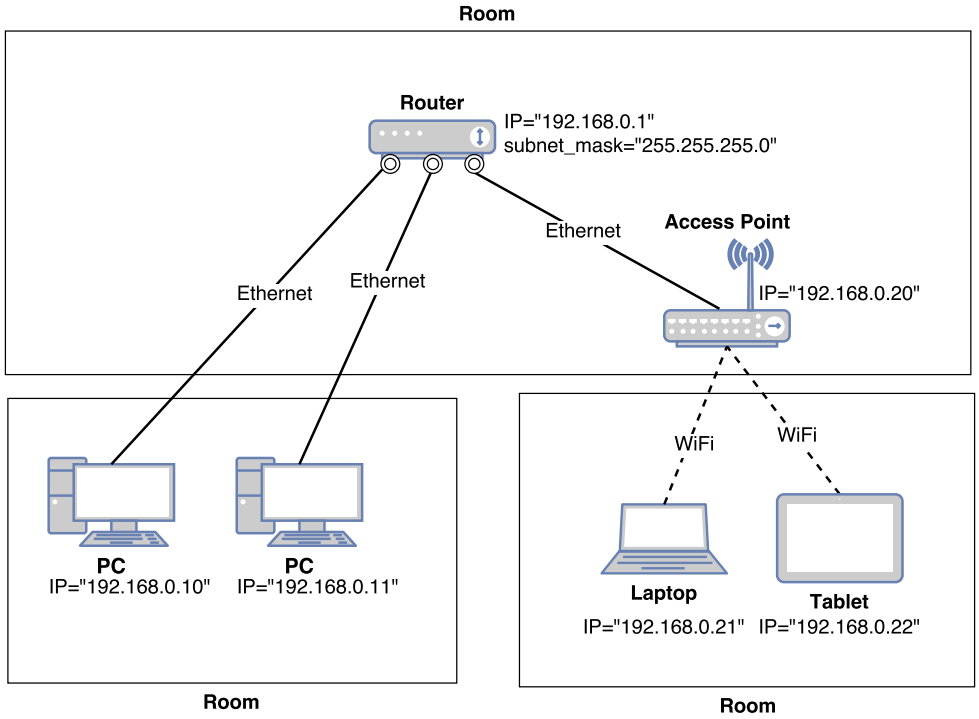


Figure 2.3: Complex example model with spatial relationships

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 2.3 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 and electronic whiteboards. While also featuring sketch recognition of shapes in a lexicon, it does not require an explicit learning phase: as soon as a drawn shape can not be recognized, the user is asked to either manually pick a type from the lexicon or introduce a new type for the shape. The sketch recognition itself is again based on a Levensthein distance comparison [121]. Furthermore, shapes can be also be annotated with text as shown in figure 2.2. However, such annotations are purely visual information and have no influence on the actual metamodel. Spatial relationships are not supported, thus making it impossible to input any example models such as the one depicted in figure 2.3.

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, including *yED*. Such editors typically come with a set of predefined icons for various domains and also support the import of custom icons. Therefore, it is possible to sketch example models with the complexity of 2.3. The tool also 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 [98], 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 2.3.

The results of the analysis of the unconstrained input are summarized in table 2.2. *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 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 [86]. Furthermore, for the sketch recognition algorithm to work effectively, hand-drawn shapes need to be sufficiently distinguishable from each other. This restriction is not present when using an icon-based language of annotated vector graphics as done by for example *metaBup*.

2.2.2 Metamodeling Support

A metamodel captures the abstract syntax and the static semantics of a DSML and constrains the structure of the language elements. One of its purposes is to define which models belong to the language. A model that belongs to a language constrained by the metamodel is said to *conform* to the metamodel. Therefore, a key aspect of example-driven DSML design is to infer conformance rules from a set of example models. Typically, this is done by generating a metamodel from these example models. Ideally, such a metamodel inference algorithm can use

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 2.2: Tool support regarding example model input capabilities

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 generation algorithm must be able to handle all of these potential issues.
Metamodel inference is also used in areas outside of DSML design. For example, *MARS* [52] 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 unambitious.

Beside imposing structural constraints to the language, metamodeling support in the scope of example-driven DSML design can be defined by the following aspects:

1. Is it possible to perform MDE activities such as code generation, model analysis or simulation to the instance models?
2. If yes, can such activities be performed in the same environment, or must an additional export step precede?
3. Lastly, can the example models be reused as instance models and be subject to the same MDE activities as the instance models?

To evaluate how language constraints are generated from the example models, the same test models that were presented in paragraph 2.2.1 are used. Obviously, the overall complexity of a generated metamodel (if one is generated in first place) is directly linked to the expressiveness of the example model: if a tool only supports shapes and links as depicted in example model 2.1, 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 2.2 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 and required user input is taken into account as well. The other factors are assessed based on the information given in the literature.

1. *Scribbler* requires the user to externally define the metamodel of the DSML within the Eclipse Modeling Framework (EMF) as an Ecore model [7]. 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 [8]. 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. However, due to the maturity and wide-spread use of the EMF as a metamodeling framework [106], advanced metamodels can be built. Furthermore, since all models of *Scribbler* can be exported to the EMF, all MDE activities that are supported by this environment can be used.

2. *MLCBD* supports the semi-automatic inference of the DSML metamodel. Proceeding from the recognized sketches, it defines an algorithm that incrementally builds the metamodel with user interaction where necessary [19]: 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. is a link 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 [18]. In his PhD thesis, Cho extended the described approach with model space exploration [17]: After the metamodel is successfully inferred, a set of 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. 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. MDE activities are not supported, since the tool is implemented as a self-contained application within the drawing and diagram application *Visio*. Additionally, no evidence was found that models can be exported to other metamodeling environments.

3. *FlexiSketch* combines the freehand sketching aspects of *Scribbler* with the hidden metamodel construction of *MLCBD*. 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 can select their types from a set of given options such as unidirectional or bidirectional as well as their visual representation (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. The metamodel is incrementally built in the background while the user sketches. By locking the metamodel, it can no longer be updated through any sketching activity. In this mode, the tools essentially becomes a metamodeling environment where instance models can be drawn the same way as example models. However, the metamodel generation does not support advanced constructs such as inheritance, abstract classes or compositions. Since the tool is implemented as a standalone application, it does not feature any support for MDE activities. Also, it does not seem to be possible to export model artifacts from the tool. Therefore, no metamodeling support beyond inferring basic language constraints is supported.

4. *metaBup* is capable of automatically deriving an explicit metamodel which is transparent to the user and can be manipulated. Starting from the sketches, an importer generates a textual fragment that describe the classes, links and hierarchies. This importer first converts the sketch to a model that conforms to the sketch metamodel of *metaBup*. 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. Particularly, spatial information about overlapping shapes that is present in the example model gets 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 viewed and edited by the user. 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 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.
Since *metaBup* supports detection of spatial relationships, a metamodel derived from example model 2.3 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.
Through its integration in the EMF, *metaBup* features extensive metamodeling support: since the actual modeling of language instances is performed in the EMF itself, all instance models can be subject for typical MDE activities without the need to export them first. Furthermore, example models can be reused as instance models as well by importing them into the modeling environment.

5. Unlike the other discussed tools, the *Model Workbench* focuses on example-driven DSML design where the input models and the generated metamodel are of textual nature. 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: 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 [99]. 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 [17], 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 introducing inheritance and enumerations. 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.
Due to the fact that no literature about the *Model Workbench* as a metamodeling environment seems to exist, assessing the support for MDE activities within the environment is not possible. Although the *Model Workbench* is advertised as a modeling platform for the definition and use of modeling languages, no resources could be found that prove support for model transformations, which serve as the backbone for many MDE activities.

The features and capabilities of each tool regarding the metamodeling capabilities and the metamodel inference process in particular are summarized in table 2.3. *metaBup* and the *Model Workbench* explicitly derive 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 modeling activities.

2.2.3 Co-Evolution

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

Tool	Language Constraints	Advanced Meta-Constructs	Automation	MDE activities
Scribbler	Manual metamodel definition	None	Manual	Supported by EMF
MLCBD	Implicit metamodel generation	None	Semi	None
FlexiSketch	Implicit metamodel generation	None	Full	None
metaBup	Explicit metamodel generation	Inheritance, abstract classes, compositions	Full	Supported by EMF
Model Workbench	Explicit metamodel generation	Inheritance, abstract classes	Semi	Not assessable

Table 2.3: Tool support regarding the metamodeling capabilities of each approach

[80]. 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 [46]. In their recent survey, Hebig et al. give an overview of 31 different approaches that investigate metamodel co-evolution [47]. 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 [82]. Lastly, Gruschko et al. [40] have classified metamodel changes by their effect on the conformance relationship. Hereby, changes to metamodels can be either *non-breaking*, in which case no instance models break, *breaking and resolvable*, where instance models break, but the conformance can be repaired by automated means or *breaking and unresolvable*, where instance models breaks and cannot be repaired by automatically. This taxonomy is nowadays widely used to categorize metamodel changes.

Cho et al. identify the co-evolution of models as one of the key challenges in example-driven DSML design [20]: Language evolution becomes particularly interesting because only rarely, all language requirements are met or even known within the first development iteration. 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 models and metamodel diverging and therefore breaking the conformance relationship between example models, instance models and the 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 section discusses the different approaches to model co-evolution present in the investigated tools. To assess the model co-evolution capabilities of every tool, we differentiate between the following two evolution scenarios:

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.

To systematically assess the co-evolution capabilities of each tool, test cases for each scenario were designed. These test scenarios are divided into forward- and backward evolution, but also evaluate the level of automation and existence of a systematic classification. Therefore, test cases for both breaking and resolvable and breaking and unresolvabe changes exist for both directions. Non-breaking changes were not assessed, since they do not break any models and thus have no significance for the co-evolution problem.

Using the sample language that was already utilized for the models in section 2.2.1, the test cases for backward evolution are given in figure 2.4 and 2.5, respectively. The resolvable change renames the metaclass "Router" to "DSL-Router", effectively evolving the language. As a consequence, neither the example models nor the instance models conform to the new metamodel anymore and therefore need to evolve as well. The change is resolvable since the same rename operation can be automatically applied to the models to repair the conformance relationship. The unresolvable change adds the mandatory metaclass "Switch" to the metamodel and breaks example- and instance models, since a router element cannot be connected to a PC any longer. This change is considered as breaking and unresolvable because there exists no unambiguous method to automatically instantiate and connect the new class in all models. These two scenarios can be transferred to forward evolution as well. Figure 2.6 shows how a class instance is renamed in all example models. Although no classification of changes for example models seem to exist in the literature, the change is resolvable since the metaclass can be renamed accordingly in both the metamodel and the instance model in an automated manner. Analogous, figure 2.7 shows the addition of an obligatory class instance to both example models. As a result, the metamodel does not reflect the example models anymore. Although automatically adding the new class to the metamodel might be possible, the instance model has to evolve as well, making this change unresolvable without manual intervention.

The remainder of this section investigates if and how the different tools support the two co-evolution scenarios. Since only *metaBup*, the *Model Workbench* and *Scribbler* expose the meta-

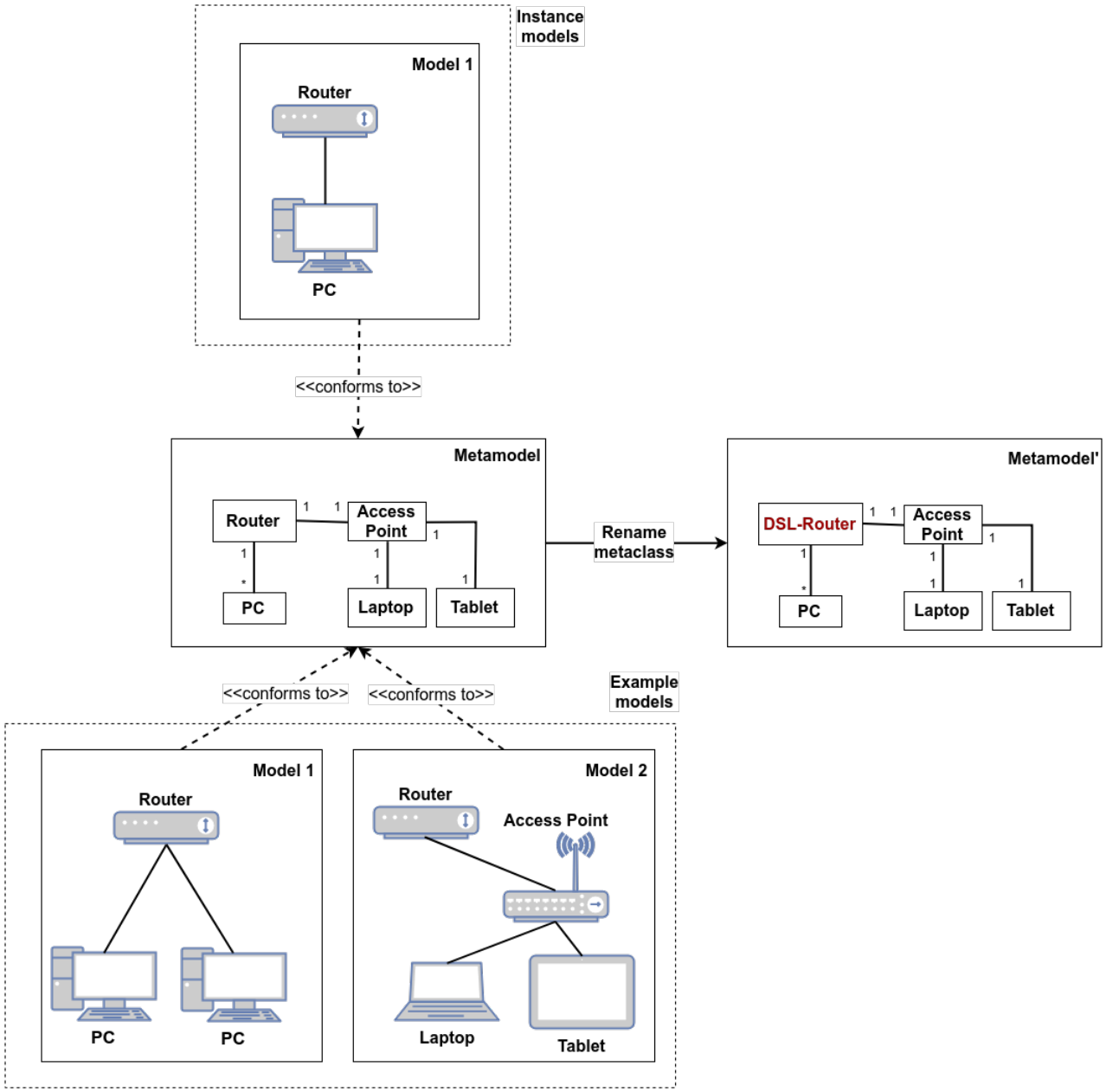


Figure 2.4: Resolvable backward evolution test case: A metaclass is renamed

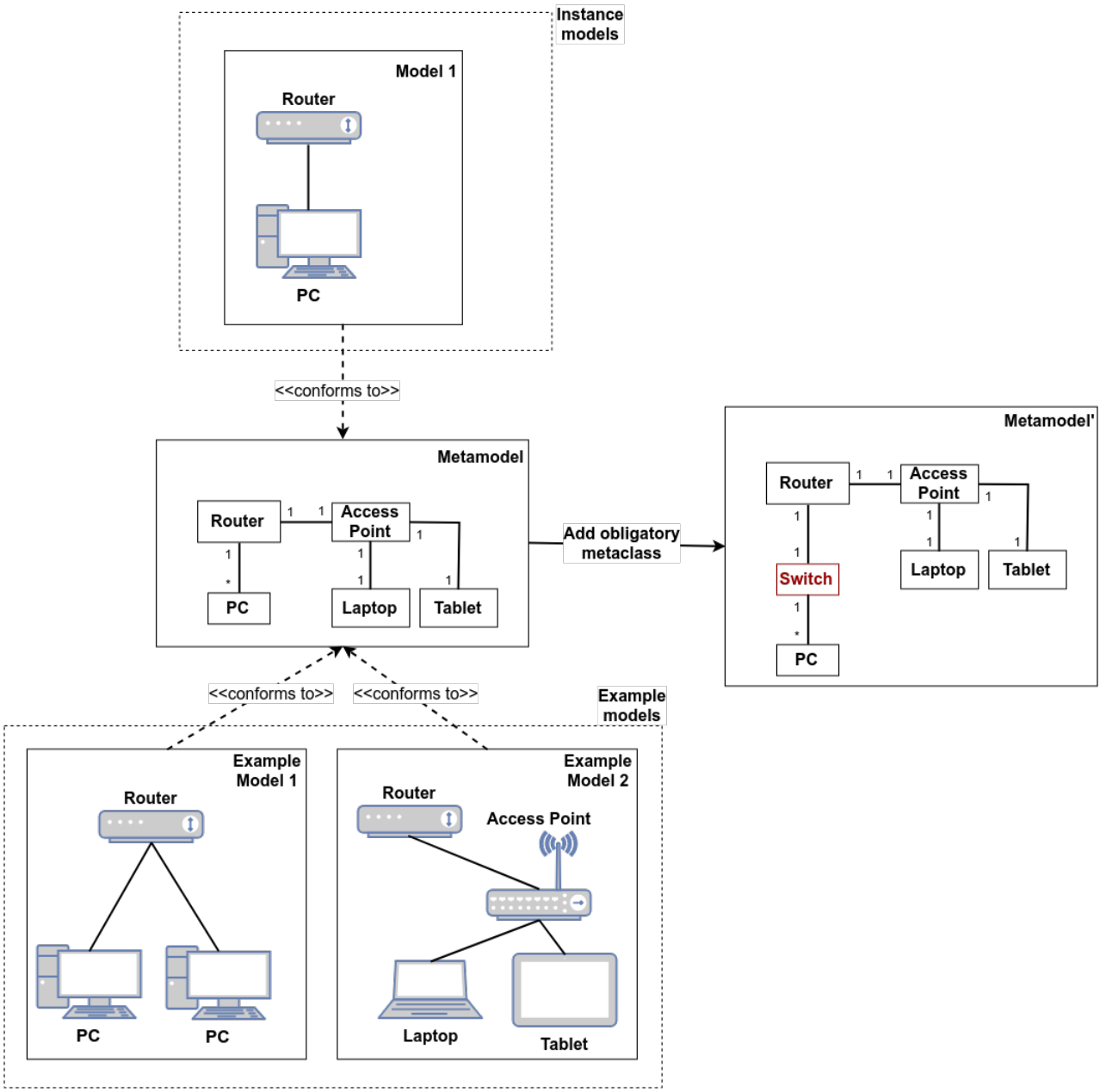


Figure 2.5: Unresolvable backward evolution test case: An obligatory metaclass is added

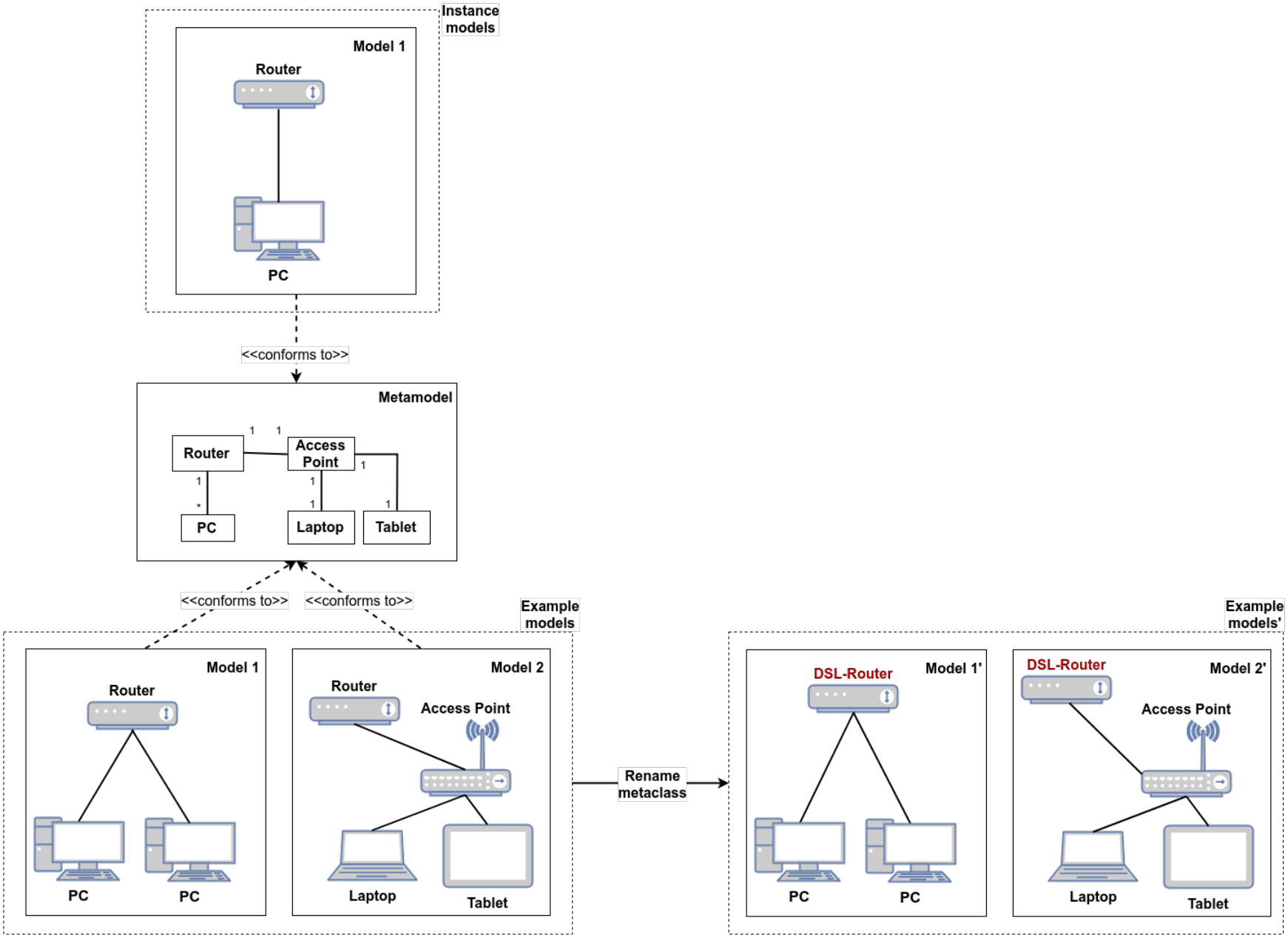


Figure 2.6: Resolvable forward evolution test case: A class instance is renamed in all example models

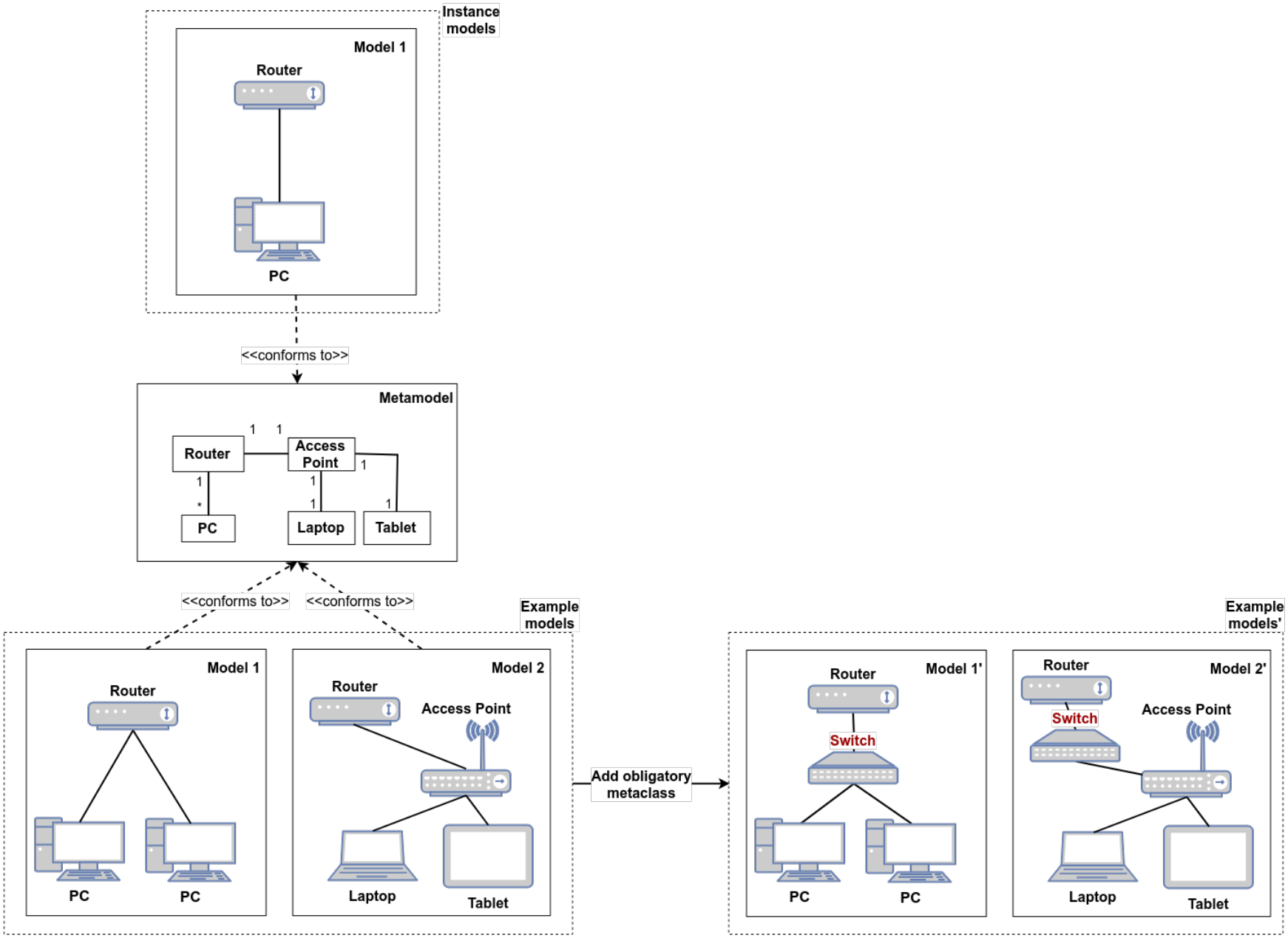


Figure 2.7: Unresolvable forward evolution test case: An obligatory metaclass is added to the example models

model to the user (see section 2.2.2), 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 scenarios depicted in figure 2.4 and 2.5, the user has to manually retain the conformance relationship by renaming the example model elements and introducing the new "Switch" element, respectively. Furthermore, no systematic classification of evolution scenarios could be found in the literature.

2. *MLCBD* supports forward evolution by re-iterating the metamodel inference process. In both scenarios 2.6 and 2.7, after the example models are changed, the metamodel can be updated by re-generating it. However, no information is given about if and how existing instance models are co-evolved when the metamodel is re-generated. Furthermore, no classification of possible evolution scenarios could be found. Lastly, backward evolution is not supported due to the metamodel being generated implicitly only and therefore not being exposed to the user.

3. *FlexiSketch* does not expose a metamodel to the user and therefore only supports forward evolution. The tool mitigates issues during forward 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. Merging metamodels with breaking changes does not seem to be possible. Second, a lock mechanism can be triggered by the user that disallows any modifications to a metamodel. Therefore, instead of evolving the model, non-conforming parts of an example model will be highlighted. When considering the resolvable forward evolution scenario 2.6, *FlexiSketch* maintains one metamodel for every example model. Therefore, a resolvable change, such as the depicted renaming of a class instance automatically evolves the underlying metamodels. The same holds true for unresolvable changes, such as the one depicted in 2.7. However, no information is given on how instance models that conform to the generalized metamodel can be evolved, especially after the lock mechanism is removed and the language evolved by changing example models.

4. *metaBup* supports metamodel refactorings of the generated metamodel [67][66] and must therefore provide a mechanism for backward evolution. The refactoring process is guided by a visual assistant that provides several predefined 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 [31] and Demeyer et al. [27]. The approach

follows a taxonomy described by Gruschko et al. [40]: metamodel changes are classified into *non-breaking, breaking and resolvable,* and *breaking and unresolvable.* The solution supports automatic updating of the imported example model fragments (but not the sketched example models since they reside in an external drawing tool) 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 fragments 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 and re-generating the metamodel. 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.
In *metaBup*, the generated metamodel is used to automatically generate a modeling environment using Eclipse Sirius. In this environment, the user can instantiate instance models that conform to the language. However, at this point it is difficult to evolve the metamodel by changing the example models, since the environment then needs to be re-generated and instance models need to be migrated manually.
Considering the test cases for backward evolution, a resolvable change to the metamodel as shown in 2.4 is supported since metamodel refactorings are classified and supported. However, only the textual example model fragments do evolve, but not the example models themselves, since they were generated in an external drawing application and only exist as a collection of image files for *metaBup*. Unresolvable changes, such as the one depicted in 2.5 raise an issue and the user is asked to manually repair the example model fragments.
Forward evolution is supported as long as no metamodeling environment has already been generated from the metamodel. By providing an updated set of example models, *metaBup* can re-generate the metamodel to reflect the changes. During this process, contradicting example models are detected and issues can be raised, thus requiring user intervention. Therefore, both 2.6 and 2.7 are supported by re-generating the metamodel. However, if an instance model has already been created using the old metamodel, it must be manually migrated to the new metamodel.

5. The *Model Workbench* supports co-evolution features comparable to *metaBup*. For forward evolution, resolvable changes such as the rename in figure 2.6 are applied automatically upon metamodel regeneration. For every other changes, the user is asked to resolve them manually. Backward evolution supports non-breaking changes only. For both scenario 2.4 and 2.5, the tool notifies the user about the broken conformance relationship

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 2.4: Summary of co-evolution features

between the modified metamodel, the example models and the instance model.

Table 2.4 summarizes the co-evolution support of the tools. *metaBup* and the *Model Workbench* feature the most complete implementation and use the commonly found taxonomy of diving the evolution scenarios in non-breaking, breaking and resolvable and breaking and unresolvable at least to some extend. In every case, if a change is unresolvable, the user is notified and asked to resolve it manually.

2.2.4 Tool Support

The quality of a tool is assessed in multiple dimensions: **implementation**, **integration**, **usability** and its **collaboration** support. The results are shown in table 2.5.

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 choices. For example, while *metaBup* emphasizes metamodeling and DSML design aspects and therefore is implemented as a plugin to Eclipse, *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

	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 2.5: Summary of tool support evaluation

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 [17] 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)[7] 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 example-driven DSML design, 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

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

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 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 example-driven DSML design since, typically, the process 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* [123] 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.

2.2.5 Comparison

Figure 2.8 shows the results of the evaluation as a radar chart. It points out the maturity of each solution in the individual areas *Unconstrained Input, Metamodeling Support, Co-Evolution* and *Tool Support*. For the rating, a point-based system was used, based on the previous analysis. Section 2.2.1 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 2.2.2 analyzed the metamodeling support of each tool in various dimensions. Weak support signifies only basic support for metamodeling activities, where for instance models first have to be exported to a different environment to make use of MDE activities. On the opposite, good support means that the whole approach is integrated and build around MDE principles. This includes the reusability of exam-

ple models and the equal access of MDE activities such as transformations to all model artifacts. Section 2.2.3 compared the co-evolution support of the tools. A distinction between forward evolution backward evolution was made. Furthermore, it was investigated if a classification of changes exist and how the approaches handle test scenarios that are considered resolvable and unresolvable. Weak support in this area means that no proper classification exists and the tool is not capable of handling any test cases while advanced support indicates the presence of a systematic classification and full co-evolution support among all model artifacts. Lastly, section 2.2.4 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.

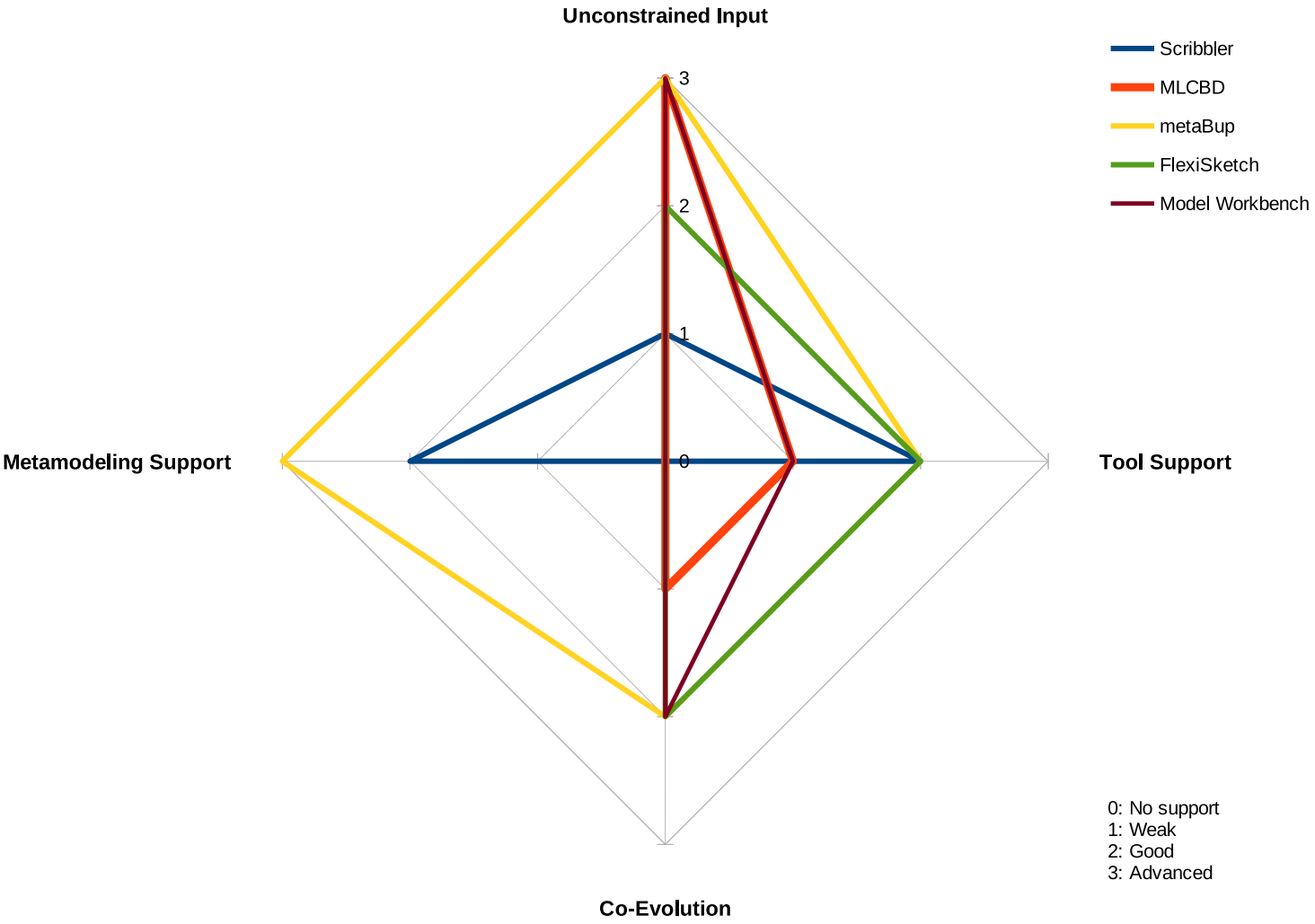


Figure 2.8: Summary of strengths and weaknesses of analyzed solutions

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 example-driven DSML design 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.
- 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 modeling process, do not expose the metamodel to the user. Advanced metamodeling support is only present in *metaBup*, which allows the instantiation of models in a generated metamodeling environment based on the Eclipse framework and additionally allows to reuse example models.
- Support for model co-evolution is rather diverse, spawning from no support at all to intermediate levels. Naturally, a correlation between metamodel generation and evolution maturity exists: if the metamodel can not be manipulated by the user, backward evolution becomes impossible. Furthermore, not all solutions perform a systematic classification of possible evolution scenarios. No solution does support evolving the language by changing the example models once instance models have been created.
- Mostly being research prototypes, the tool support itself was the only area where no tool achieved an advanced level. While all approaches were implemented as either plugins or standalone applications, most of them are unavailable, making a throughout evaluation difficult.

2.3 Conclusion

This chapter 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, metamodeling capabilities, evolution of the DSML 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.
- Tools either impose language constraints by either explicit or implicit metamodel generation. The advantages and disadvantages of both approaches have not been systematically investigated yet, thus permitting a statement about one approach being generally superior to the other.

- Concrete syntax is seldom touched topic, as it is implicitly defined by the example models. Most solutions force the language designer to specify a single shape for each meta-concept. Therefore, the mapping between concrete and abstract syntax is mostly manually defined as a one-to-one relationship. As a consequence, a concrete syntax representation for a concept is difficult to alter once it has been introduced.
- 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 example-based 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 example-driven DSML design 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 at the scope of large projects exist, suggesting that the approach has not attracted attention in the industrial context yet.

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

Chapter 3

Theoretical Concepts

In this chapter, the theory of our approach gets explained in detail. First, section 3.1 frames the chapter by introducing requirements that will serve as goals for our approach. Then, section 3.2 presents individual key aspects of the approach that are used to fulfill one or more of the previously defined requirements. In particular, it describes how example models can be elevated to the primary descriptive element of a modeling language by defining a conformance relationship between instance and example models. Furthermore, it discusses all operations that can be performed on both example- and instance models by defining a common graph-like data structure for these model artifacts. Section 3.3 shows how the problem of language evolution can be solved by performing language changes on the level of example models only. Section 3.4 elaborates on the role of the concrete syntax and presents an approach to store the representation of model artifacts independently from the conceptual models, thus improving the agility of the design and making it possible to evolve the concrete syntax together with the abstract syntax. Lastly, 3.5 consolidates the previously presented concepts to an agile process for integrated language design by example.

3.1 Introduction to Moodling

One of the main shortcomings of the existing example-driven DSML design approaches that were investigated in section 2 is the strict separation of the language design and usage phases. This separation is not only present on a conceptual level, but also manifests itself in the use of different tools for each phase. For instance, general-purpose drawing tools are used to sketch example models, which are then imported in a meta-modeling environment. This separation impedes language evolution driven by changing the example models, since they reside in a different environment than the metamodel and the instance model. Other solutions implement a stand-alone application that build a metamodel during sketching and thereby iteratively

constrain the user. However, such solutions rarely implement any metamodeling features such as model transformations or process modeling, as they focus on the sketching aspect. As a result, they require exporting the metamodel to metamodeling environments to support the aforementioned activities. Either way, model artifacts are separated between different tools or phases, resulting in a disintegrated design process with a high cognitive load. Furthermore, it makes language evolution complex, since almost always, either the example models or the instance models diverge from the evolved language and need to be updated manually.

We therefore see the need for an example-driven DSML design process which adopts the agile principles of incremental and iterative software development. We propose the name *Moodling* for our envisaged approach as a blend of words between "Doodling" and "Modeling". This emphasizes the aspect of combining a fast and agile language design process with established metamodeling fundamentals. Since one main aspect of our process is the integration of all available activities in a single environment, we do not explicitly generate a metamodel and expose it to the user, as this would have multiple negative implications: first, it would increase the cognitive load of the whole process, since language design would be possible on the level of instances (the example models), but also on meta-level (the metamodel). This entails the risk to confuse the user, especially if they have no experience of metamodeling concepts. Second, exposing a metamodel to the user increases the number of artifacts that need to be considered during co-evolution, since language changes could be triggered by either changing the metamodel or the example models. While existing solutions partially solve this by re-generating the metamodel in case the example models have changed or co-evolving instance models in case the metamodel was adapted, both approaches force the user to manually migrate either the instance models (when a new metamodel was generated) or the example models (when the metamodel was directly manipulated).
Implicitly generating a metamodel reduces the cognitive load but does not solve the co-evolution issue: language changes are triggered by adapting the example models, but still requires to co-evolve the metamodel as well as the instance models. Even more, an implicit metamodel can be seen as merely an intermediate representation of the example models with the purpose to impose restrictions on further modeling activities. For these reasons, we propose the use of example models as first-class entities to describe the DSML throughout the whole process. To capture the implications of this decision and to frame the theoretical foundations presented in this chapter, we formulate the following key requirements for Moodling that we believe are required to eliminate the shortcomings of existing approaches:

R1: Throughout the whole process, the example models are the primary artifact to describe the language's abstract and concrete syntax. This is possible since we aim for an integrated approach where all artifacts reside in the same modeling environment.
R2: As a consequence of R1, language evolution can only be driven by changing the example models. This includes both the abstract and the concrete syntax.
R3: Example models are full-featured models on their own and, on a conceptual level, do not

differ from instance models. In particular, example models can be reused as instance models.
R4: To gain immediate feedback over the adequacy of the language, the feedback loop between designing and using the language must be as short as possible. This also enables fast reaction to changing requirements.
R5: To reduce the cognitive load and make DSML design accessible for non-experts, meta-modeling activities must be hidden from the user whenever possible. This is also reflected in R1 and R2, since the only exposed artifact are example models.

These requirements will be used as a foundation for the remainder of this chapter to reason about design decisions and referenced to whenever an introduced concept helps to meet the respective requirement.

3.2 Elements and Activities

Although the goal is a single, integrated approach to an example-driven DSML design process that fulfills requirements R1-R5, the process consists of various elements which, combined, will form the building blocks for our approach. The following subsections subsequently introduce these individual elements of Moodling before they are combined into an actual process.

3.2.1 A Common Metamodel

One key element of our approach as well as a criterion for requirement R3 is a data structure that can describe the structure of both the example and instance models. Generally, it can not anticipated what kind of language the user wants to design. For that reason, the data structure must be as generic as possible and be able to capture a wide range of model properties. Within the MDE community, there is general consent that even complex systems can be modeled by means of graphs, which provide a powerful, yet intuitive approach with a strong mathematical foundation [79]. In particular, graphs can be manipulated by graph rewriting techniques, which are a fundamental concept for model transformations [39]. In fact, many of today's modeling languages are graph-based in their nature. Examples for such languages include Petri nets, statecharts, and activity diagrams. It was also noted that many initial sketches that are used to reason on a new project are intrinsically graph-like [5].
For these reasons, a graph structure was chosen as a common descriptor for all models in the approach. We believe that a graph is capable of capturing all essential aspects of the language to design and provides means to formally reason about certain model properties. Besides nodes and edges, the graph must posses a notion of types, of which nodes (and possibly edges) are instances of. This relationship is comparable to the relation between objects and classes in object-oriented programming. Furthermore, nodes (and, again, possibly edges) can have attributes to capture certain properties of a specific instance. In object-oriented programming, such attributes are

commonly described by a name and a type the value of the attribute has to conform to. Graph structures with types and attributes do exist in the form of so-called attributed type graphs *(ATG)*. ATGs provide a powerful primitive to model real world entities and their relations and are a well-researched domain with a strong mathematical foundation [29][97]. We define an attributed type graph within the context of Moodling as follows:

Definition 1 (Attributed type graph)**.** *Let T be a non-empty set of types and L be a set of labels. We define $G = (V, E, T, A)$ as an attributed type graph (ATG) with node set $V \subseteq L \times T$, edge set $E \subseteq V \times V$ and a set of attributes A of cardinality $\mid V \mid$ with ω-dimensional attribute vectors $a_i \in A$.*

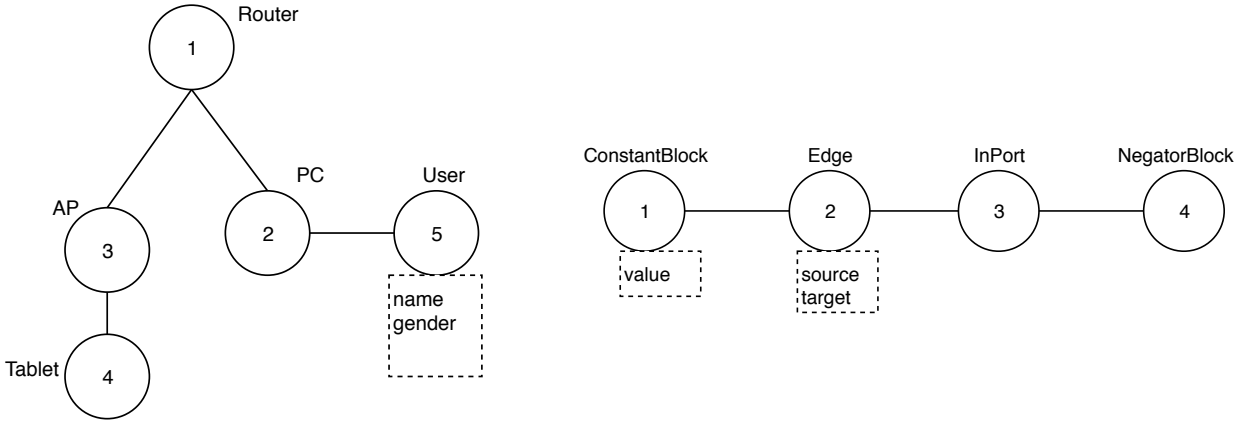
We denote nodes $(\upsilon_1, \tau_1) \in V$ by $\upsilon_1 : \tau_1$ and say that the node with label $\upsilon_1 \in L$ is *typed* by $\tau_1 \in T$. We furthermore denote edges by $\{\upsilon_1 : \tau_1, \upsilon_2 : \tau_2\}$. Note that two distinct nodes $\upsilon_1 : \tau_1, \upsilon_2 : \tau_2 \in V$ with $\upsilon_1 \neq \upsilon_2$ can have $\tau_1 = \tau_2$ (i.e. can have the same type). The ω-dimensional attribute vectors $a_i \in A$ are paired with their corresponding nodes $\upsilon_i : \tau_i \in V$ and represent attribute identifiers of that particular node. For instance, if $\upsilon_i : \tau_i \in V$ represents a person, then the associated attribute vector a_i of dimension $\omega = 2$ could contain two values with a_{i_1} as "name" and a_{i_2} as "gender". Note that for the remainder of this chapter, we assume the elements a_{i_j} of every attribute vector $a_i, i = 1, ..., |V|$ to be unique, i.e. $\nexists k, l \in \mathbb{N} \mid a_{i_k} = a_{i_l}$.

Both the example and instance models in Moodling will be expressed as attributed type graphs and therefore do conform to the same metamodel which describes such an ATG. A common metamodel is possible under the premise that all models reside within the same modeling environment which holds the ATG metamodel. Note that the definition of an ATG does neither define a type system nor attributes for edges, which can be emulated by modeling the edge as a typed and attributed node.

Figure 3.1 shows an example of two ATGs, using the common representation for graphs with circles as nodes and lines as edges, where the nodes are annotated with their labels, types and optionally attributes. While the left subfigure describes the structure of a home network from the same DSML already used in chapter 2, the right subfigure depicts a Causal Block Diagram (CBD). CBDs are a common formalization for blocks and connections which can be used to describe the relationship between input- and output signals in physical systems [37]. The CBD example shows the previously mentioned limitation of our ATG, where edges are neither typed nor attributed and always unidirectional. For that reason, a directed edge is modeled using a node of type "Edge" with source and target attributes.

ATG transformations

Graph transformations are the underlying concept for model transformations. Model transformations allow, among others, for model manipulation, analysis, execution and code generation.



(a) An ATG describing a home network

(b) An ATG describing a CBD

Figure 3.1: Example of two ATGs

Since all our models are describable by an ATG, we can define all model manipulation operations using rewriting rules for graphs. Such rules are also called transformation rules [104]. Generally, they consist of a matching graph, commonly referred to as the *left-hand side* (LHS), and a replacement graph, called the *right-hand side* (RHS), which replaces the matching LHS. Additionally, if a preceding *negative application rule* (NAC) also matches the graph, the transformation is not executed. Figure 3.2 shows an example of such a rule in the established concrete syntax. The pink numerical annotations are used to link LHS elements to elements of the RHS. As long as the LHS is found in the graph, the transformation rule is applied. In this case, every node typed by "Router" is connected to a "PC" node, if such a link does not exist already. Finding a LHS match in the graph requires to solve the subgraph isomorphism problem, which is considered to be $\mathcal{NP}$-complete [26]. Therefore, no efficient algorithm for matching an isomorphic subgraph exists. State-of-the-art algorithms are capable of handling graphs with up to a few thousand nodes [125]. However, since the graphs in the presented approach are usually created by hand, they are unlikely to exceed a size where the performance of subgraph matching becomes an issue.

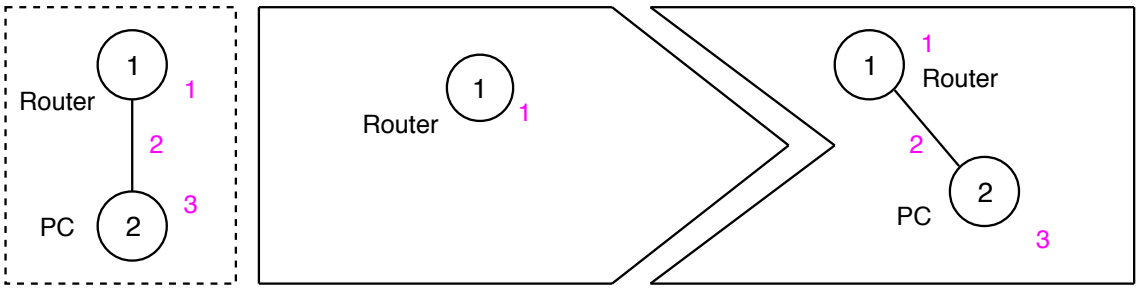


Figure 3.2: Example of a transformation rule with NAC, LHS and RHS

The LHS, RHS and NAC patterns of a transformation are models of a language which can be derived automatically from the metamodel the models that are subject to the transformation conform to. This approach was introduced by Kühne et al. [60] and proposes to *Relax*, *Augment* and *Modify (RAMify)* the metamodel in order to arrive at a modified version of the original language which is suitable to model the patterns of transformation rules.

3.2.2 Model Consistency and Conformance

In metamodeling, every model has to conform to another model. Usually, such a conformance relationship exists between instance models and metamodels, but also between metamodels and meta-meta-models. For example, the *Meta Object Facility* (MOF) describes four abstraction levels M0-M3, whereby M3 is the meta-metamodel which is capable of describing metamodels and additionally can describe itself. This property is sometimes referred to as *metacircularity*. M2 contains metamodels, which are instances of the M3 model. M1 contains instances of M2 models and M0 holds actual real-world data [92].

The definition of the conformance relationship between instance models, metamodels and meta-metamodels has been extensively covered in the literature [59][58], with recent research going towards explictly modeling the conformance relationship to increase flexibility [112]. However, one key aspect of our presented approach is the use of example models as primary description of the structure of the language. In that sense, it differs from various existing approaches presented in chapter 2, where a static metamodel is generated from the set of example models. There, the generated metamodel captures a set of constraints that each instance model has to satisfy in order to be a valid instance of the metamodel. For UML class diagrams, such well-formedness constraints include element multiplicities or attribute and association typing. For our approach however, we decided against explicitly generating a metamodel from the example models for the following reasons: First, a metamodel would serve as an intermediate proxy element between instance and example models and therefore violate requirement R1, since the metamodel would become the primary artifact to describe the language structure. Second, if such a metamodel is exposed to the user, it becomes possible to perform language evolution on metamodel-level. This conflicts with requirement R2, which states that language evolution should only be possible by changing the example models. Lastly, an explicit metamodel increases the amount of artifacts involved in the process and therefore both increases the cognitive load (R5) and the length of the feedback loop (R4) since an extra metamodel generation step is required.

In Moodling, all constraints, which together decide about the well-formedness of an instance model, are inferred directly from the example models. There is hence a need for a definition of such constraints and for a method to verify the consistency of an instance model against a set of example models using these constraints. For our approach, we investigated the following structural language properties which can be constructed from a set of example models described by ATGs:

1. Node typing
2. Type cardinality
3. Attribute typing
4. Attribute cardinality
5. Edge typing
6. Edge cardinality

The constraints derive their origin from the linguistic conformance check between metamodels and instance models as it can be commonly found in metamodeling tools [113]. They will be used to define the conformance relationship between an instance model and the example models, i.e. define a set of rules to decide whether an instance model conforms to example models. The next paragraphs elaborate on each of these properties and present constraints that the instance model has to fulfill. Hereby, we assume the existence of an an instance model $G_{im} = (V_{im}, E_{im}, T_{im}, A_{im})$ and a set of example models $S_{xm} = \{G_{xm_1}, G_{xm_2}, ..., G_{xm_n}\}$ of cardinality $n \in \mathbb{N} > 0$ with $G_{xm_i} = (V_{xm_i}, E_{xm_i}, T_{xm_i}, A_{xm_i}), i = 1, ..., n$.

Node Typing

Every element in an instance model must be typed by an element on meta-level. In object-oriented programming, this corresponds to verifying the type of objects with regard to the available types. In Moodling, type information is available from the typing of a node. Therefore, the node typing constraint for an instance model can be defined as follows:

Definition 2 (Completely typed instance model)**.** *An instance model G_{im} is completely typed by S_{xm} if*

$$T_{im} \subseteq \bigcup_{i=1}^{n} T_{xm_i}$$

On other words, for every node in the instance model, there must be at least one node with the same type in any example model. If that constraint is violated, the instance model does contain one or more incompletely typed nodes and is therefore invalid.

Type Cardinality

At metamodel-level and using class diagrams, the cardinality of class instances is usually defined by additional constraints associated with the class, where a lower bound denotes the minimum number of objects and the upper bound the limit of the maximum objects. A lower bound of 0 implies an optional object that does not have to occur and a $*$ signifies an unlimited number of possible occurring objects.

For ATGs, we define the node cardinality attribute based on the node typing information:

Definition 3 (Type cardinality)**.** *Given an ATG $G = (V, E, T, A)$, the cardinality of a type $\tau \in T$ is defined as the number of nodes $v_i : \tau_i \in V \mid \tau_i = \tau$.*

In Moodling, such cardinality information needs to be inferred from the example models. Here, it is impossible to calculate the intended cardinalities without further meta-information given by the user. This is because the example models only capture the state of the system to model at a specific time as opposed to a metamodel which captures all possible states. Especially the upper multiplicity limit poses an issue since it is impossible to model $*$ and unfeasible to construct example models that reflect a particularly high cardinality of, for instance, 1000. For that reason, some existing approaches allow the user to review and manually correct the inferred cardinalities, while others do not infer such information at all and simply use the most general cardinality information $0..*$. Since asking the user to provide additional information about the intended multiplicities conflicts with requirement R5 and setting all multiplicities to $0..*$ does not infer any information at all, we decided for a compromise where the upper bound is always unlimited and only the lower bound inferred.
There are two possible ways to infer the lower bound for the type cardinality: either as binary value with 0 meaning an optional type and 1 a mandatory type or an absolute value which reflects the minimum type cardinality found for the type in all example models. In our approach, we decided for only inferring the binary value and therefore deciding if a type is mandatory or optional. The main reason for this is that essentially, example models are sketched fragments which might get created during a quick sketching session without systematical analysis and therefore should be treated with a fuzzy factor whenever possible.
To decide if the lower bound of the type cardinality is 0 (meaning optional) or 1 (mandatory), we introduce the mandatory property for types:

Definition 4 (Mandatory type)**.** *A type $\tau_m \in \bigcup_{i=1}^{n} T_{xm_i}$ is mandatory if $\tau_m \in \bigcap_{i=1}^{n} T_{xm_i}$.*

Based on that definition, we can formulate the final type cardinality constraint as follows:

> *For every mandatory type τ_m, there must exist at least one node in the instance model that is typed by τ_m. Otherwise, the instance model is invalid and does not conform to the example models.*

Attributes

In class diagrams, the name and type of every attribute of a class is defined at meta-level. To verify the attribute conformance of an instance, it is sufficient to check if the name definition and the attribute's value type correspond to the attribute definition found in the metamodel. In our approach however, attributes only exist as attribute vectors which are associated to nodes. The vector elements correspond to the attribute name in class diagrams and are unique for

every vector. Furthermore, no information about the actual type of the attributes is available, making an actual type checking of the attributes impossible. Nevertheless, it is still possible to infer two attribute constraints from the example models: first, every attribute in the instance model must have a corresponding attribute in any example model. More precisely, we define a valid instance model attribute value as follows:

Definition 5 (Valid attribute value)**.** *An instance model attribute value* $a_{i_j} \in a_i,\ a_i \in A_{im}$ *associated with its corresponding node* $v_i : \tau_i \in V_{im}$ *is valid if and only if* $\exists\ v_x : \tau_i \in \bigcup_{k=1}^{n} V_{xm_k}$ *with an associated attribute vector* a_x *that contains a value* a_{x_l} *with* $a_{i_j} = a_{x_l}$*.*

Second, an attribute value can be mandatory and therefore must occur in the instance model. Similar to the mandatory definition for typed nodes (see definition 4), a mandatory property for attributes is defined as follows:

Definition 6 (Mandatory attribute value)**.** *Let every node in* $\bigcup_{i=1}^{n} V_{xm_i}$*, that is typed by the same type* $\tau \in \bigcup_{i=1}^{n} T_{xm_i}$*, be associated with an attribute vector* $a_i \in \bigcup_{i=1}^{n} A_{xm_i}$*. Furthermore, let* S_A *a set that contains all those attribute vectors* a_i*. An attribute value* $a_{i_j} \in a_i$ *is mandatory if and only if it is element of each attribute vector* $a_i \in S_A$*.*

Based on definitions 5 and 6, we can formulate the final attribute constraints as follows:

> *Every attribute value of every attribute vector in the instance model must be valid. Furthermore, for every mandatory attribute value associated with a node typed by* τ*, there must exist an attribute vector associated with a node in the instance model, also typed by* τ*, containing the same attribute value.*

Edges

Edges define the connections between nodes, similar to associations define the connections between classes in class diagrams. In class diagrams, both ends of an association can have a multiplicity constraint, which defines the number of objects that may participate in an association. In our approach, we have already inferred object multiplicity information in 3.2.2, based on the number of typed nodes in the example models. Inferring association multiplicities based on a set of examples has been proven to be difficult, since the user's intent usually cannot be guessed [54]. Therefore, many example-driven DSML design approaches allow the user to review and refine the association multiplicities manually after they have been guessed from the examples.

Since one of the goals of Moodling is to hide all metamodeling aspects from the end-user, we decided against trying to infer exact edge multiplicities that would require a manual post-processing step. Similar to the mandatory type constraint, we set the upper cardinality to

infinity and infer the lower bound as either optional or mandatory. First, we define the type of an edge as follows:

Definition 7 (Edge type)**.** *The type of an edge* $e = \{v_a : \tau_a, v_b : \tau_b\}$ *is defined as the 2-set* $\{\tau_a, \tau_b\}$ *.*

With this definition, we can define a mandatory edge as:

Definition 8 (Mandatory edge)**.** *An edge with type* $\{\tau_a, \tau_b\}$ *is mandatory if in every example model that contains at least one node of type* τ_a *and at least one node of type* τ_b*, there exists an edge typed by* $\{\tau_a, \tau_b\}$ *for every of these nodes.*

Furthermore, we can impose the following constraint on an edge to decide whether its typing and therefore the edge itself is valid:

Definition 9 (Valid edge)**.** *An instance model edge* $e_{im} = \{v_a : \tau_a, v_b : \tau_b\} \in E_{im}$ *is valid if and only if there exists at least one edge* $e_{xm} = \{v_c : \tau_c, v_d : \tau_d\} \in \bigcup_{i=1}^{n} E_{xm_i}$ *with the same type.*

Based on the definitions 8 and 9, we can formulate the final edge constraint:

> *Every edge in the instance model must be valid. Furthermore, for every mandatory edge typed by* $\{\tau_a, \tau_b\}$*, if there exist two nodes in the instance model also typed by* τ_a *and* τ_b *respectively, there must exist an edge between these two nodes.*

3.2.3 Example Modeling

In example-driven DSML design approaches, the first step is always the generation of one or more example models. The example models do not necessarily need to reflect every aspect of the complete language (which might not even be possible due to missing language requirements or uncertainties), but can focus on singular aspects only. Since example models are ATGs, the outcome of the example modeling step is always a set of example models $S_{xm} = \{G_{xm_1}, G_{xm_2}, ..., G_{xm_n}\}$ of cardinality $n \in \mathbb{N} > 0$ with $G_{xm_i} = (V_{xm_i}, E_{xm_i}, T_{xm_i}, A_{xm_i}), i = 1, ..., n$. Note that our approach does not impose any kind of restrictions on the technique used to create the example models. In particular, it does not exclude the use of purely textual example models. However, we define a set of operations that an environment needs to provide in order to be able to effectively manipulate example models. The following paragraph elaborates on these operations.

Modeling Operations

The following modeling operations are available to manipulate an example model $G_{xm} = (V_{xm}, E_{xm}, T_{xm}, A_{xm})$:

1. **Add node:** Add a node $\upsilon : \tau$ with label υ and type τ to V_{xm}. Hereby, τ is not necessarily element of T already, i.e. can be a new type.
2. **Delete node:** Delete a node, identified by its label $\upsilon \in L$, from V_{xm}.
3. **Retype node:** Change the typing of an existing node $\upsilon_i : \tau_i$ to τ_j. Hereby, τ_j must be a new type, i.e. $\tau_j \notin \bigcup_{i=1}^{n} T_{xm_i}$.
4. **Add edge:** Add an edge $\{\upsilon_a : \tau_a, \upsilon_b : \tau_b\}$ between the two existing nodes $\upsilon_a : \tau_a \in V_{xm}$ and $\upsilon_b : \tau_b \in V_{xm}$ under the condition that no edge already exists between the two nodes.
5. **Delete edge:** Delete an existing edge uniquely identified by its two connecting nodes $\{\upsilon_a : \tau_a, \upsilon_b : \tau_b\}$.
6. **Add attribute:** Add an attribute value to the attribute vector a_i associated with the node $\upsilon_i : \tau_i$, extending its dimension w by 1.
7. **Delete attribute:** Delete an existing attribute value a_{i_j} from the attribute vector a_i, reducing its dimension w by 1.
8. **Change attribute:** Change an existing attribute value $a_{i_j} \in a_i$ to a new value.
9. **Delete model:** Delete a model $G_{xm_i} \in S_{xm}$. This includes to deleting all nodes of V_{xm_i}, all edges E_{xm_i}, all types T_{xm_i} and all attributes A_{xm_i}.

Note that none of these operations is constrained in any way. Therefore, every operation can be executed on every example model at any point in time. Furthermore, it is impossible to create contradicting example models for the following reasons: first, we do not provide a type system for attributes and edges. It therefore becomes impossible to define contradicting types for the same element in different example models. Second, the type system for the nodes is unambiguous, meaning that every node has exactly one type.

Most of the operations alter the structure of the language. As a consequence, the language can evolve and might require the co-evolution of other example- and instance models. We refer to section 3.3 for a detailed description of all evolution scenarios.

3.2.4 Instance Modeling

Conceptually, instance modeling does not differ from example modeling since instance models can be described by the same ATG and therefore the same modeling operations that were introduced in 3.2.3 can be applied to them as well. However, once the set of example models S_{xm} has been created they are used to impose constraints to the instance modeling process to maintain conformance as defined in 3.2.2. To ensure that no constraints are violated during the instance modeling phase, the available modeling operations must be constrained, that is, they

are only valid if they fulfill certain criteria. The next section introduces the available instance modeling operations alongside their respective constraints.

Modeling Operations

Given an instance model $G_{im} = (V_{im}, E_{im}, T_{im}, A_{im})$ and a set of example models $S_{xm} = \{G_{xm_1}, G_{xm_2}, ..., G_{xm_n}\}$ of cardinality $n \in \mathbb{N} > 0$ with $G_{xm_i} = (V_{xm_i}, E_{xm_i}, T_{xm_i}, A_{xm_i}), i = 1, ..., n$, we define the following constrained operations for an instance model:

1. **Add node:** Add a node $\upsilon : \tau$ with label υ and type τ. This operation is only valid if $\tau \in \bigcup_{i=1}^{n} T_{xm_i}$, i.e. there exists a node in any example model which is also typed by τ.
2. **Delete node:** Delete a node, identified by its label $\upsilon \in L$ from V_{im}. Only valid if the type is not mandatory.
3. **Retype node:** Change the type of an existing node $\upsilon_i : \tau_i \in V_{im}$ to τ_j. This operation is only valid if $\tau_j \in \bigcup_{i=1}^{n} T_{xm_i}$.
4. **Add edge:** Add an edge $\{\upsilon_a : \tau_a, \upsilon_b : \tau_b\}$ between the two existing nodes $\upsilon_a : \tau_a \in V_{im}$ and $\upsilon_b : \tau_b \in V_{im}$ under the condition that no edge already exists between the two nodes. For this operation to be valid, there must exist an edge between two nodes typed by τ_a and τ_b respectively in any example model.
5. **Delete edge:** Delete an existing edge uniquely identified by its two connecting nodes $\{\upsilon_a : \tau_a, \upsilon_b : \tau_b\}$. Only valid if the edge is not mandatory.
6. **Add attribute:** Add an attribute key to the attribute vector a_{i_j} associated with the node $\upsilon_i : \tau_i$, extending its dimension w by 1. Only valid if in any example model, there exists a node, also typed by τ_i, which is associated with an attribute a_{k_l} with $a_{k_l} = a_{i_j}$.
7. **Delete attribute:** Delete an existing attribute key a_{i_j} from the attribute vector a_{i_j}, reducing its dimension w by 1. Only valid if the attribute is not mandatory.
8. **Change attribute:** Change an attribute value a_{i_j} of an associated node $\upsilon_i : \tau_i \in V_{im}$ to a new value. This is only valid if the new attribute value is valid.
9. **Delete model:** Delete the instance model G_{im}.

In accordance to requirement R2, it is impossible to change the structure of the language during instance modeling as only instance models can be manipulated. For that reason, none of the operations requires any kind of evolution handling.

3.3 Co-Evolution

As already described in section 2.2.3, language evolution is a frequently occurring scenario where the definition of the DSML is altered to reflect changing requirements. It is of particular interest in Moodling, which focuses on the incremental and agile definition of a DSML by example. Therefore, the language is likely to be subject to frequent changes. Since no metamodel is exposed to the user, all language evolution is induced by adapting the example models to accommodate for the new language requirements. In accordance to section 2.2.3, we will refer to this as *forward evolution*.

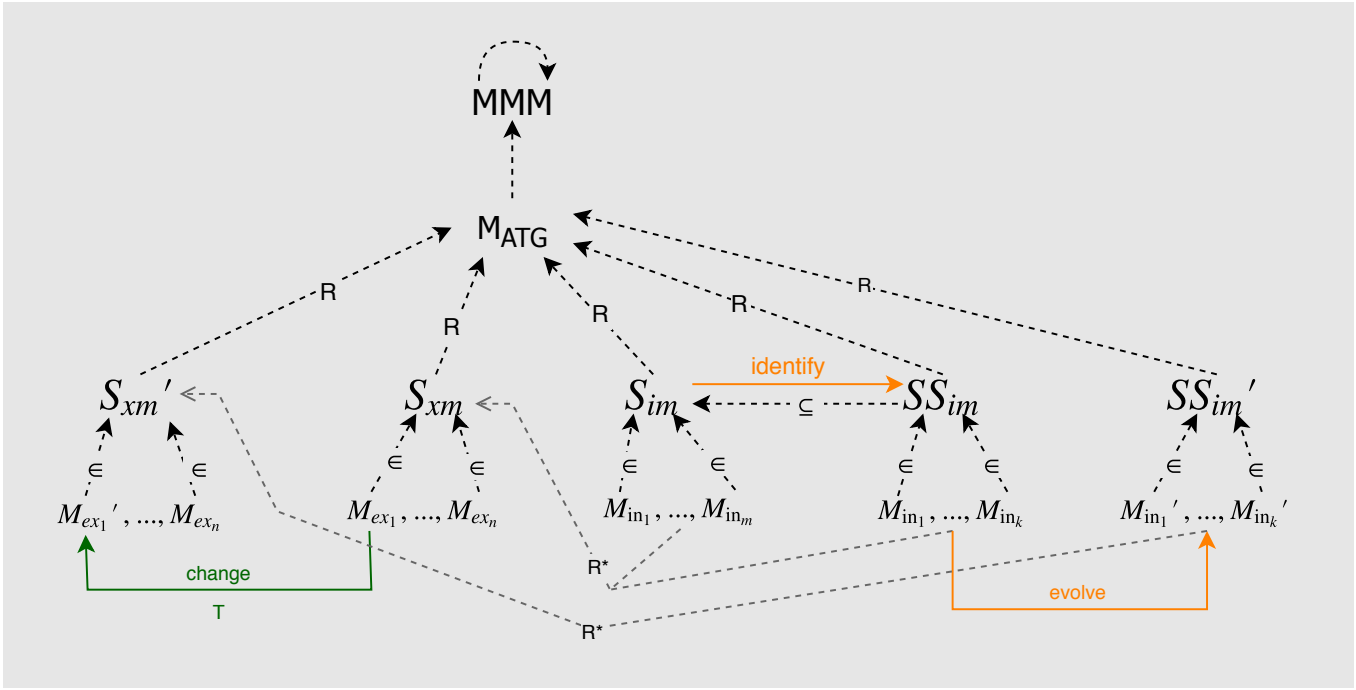


Figure 3.3: Overview of the co-evolution problem in Moodling

Figure 3.3 gives an overview of co-evolution in our approach. It shows a set of example models S_{xm} and a set of instance models S_{im}. Both of these are composed of a number of models, whereby all models in both sets conform to the ATG metamodel M_{ATG}. This conformance relationship is denoted as R. Furthermore, there exists a conformance relationship $R*$ between every instance model and the set of example models, as described in 3.2.2. If the language described by S_{xm} requires adaptations, these changes are carried out successively on the example models, resulting in a changed set of example models $S_{xm}{}'$. Such a change can be described by the model transformation T, which transforms an example model in S_{xm} to a different example model in $S_{xm}{}'$. After that transformation took place, the instance models of S_{im} do not necessarily conform to $S_{ex}{}'$ anymore, since the change could have violated one or more constraints. As a consequence, the instance models need to evolve together with the language

definition provided by the set of example models. Since not necessarily all instance models need to evolve, a subset of instance models SS_{im} first needs to be identified. Then, every instance model in SS_{im} needs to be evolved to arrive at a set of instance models that conform to the changed example models. Therefore, the main goal of co-evolution in Moodling is to maintain the conformance relationship between the example and the instance models as defined in 3.2.2 when the example models change. To achieve this, section 3.3.1 gives an overview of the terminology which is used to classify example model changes and introduces the notion of scope for every change. Then, section 3.3.2 classifies example model changes by their effects on the conformance relationship. Lastly, section 3.3.3 explains how all conformance issues in the approach can be resolved automatically.

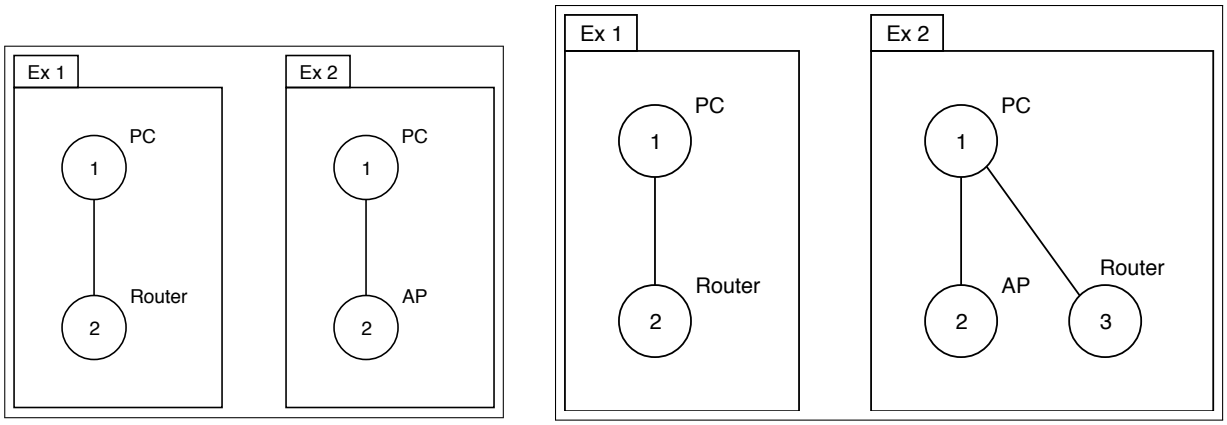
3.3.1 Terminology

Moodling allows changes that alter the structure of the language only on the level of example models. This however introduces an overhead when the structure of a language should be changed as a whole. For instance, retyping a node in the whole language requires editing every example model and manually retyping each candiate node. Depending on the amount of example models, this can be an arduous process, especially when compared to approaches that expose a metamodel (where such a change must only performed once at meta-level). To avoid this and provide a method to change all example models at once, the example modeling operations introduced in 3.2.3 can be extended with a notion of *scope*:

- The scope of a change is *global* if it is performed on all example models simultaneously.
- The scope of a change is *local* if it is performed on one example model only.

A global operation can be seen as a purely convenient method to avoid performing the same change manually on all example models. Therefore, every global change conceptually equals to performing it locally on every example model in an iterative manner. Furthermore, example modeling operations can have different *effects* regarding the conformance relationship between instance and example models. A prominent classification was introduced by Gruschko et al. [40]. Here, the authors differentiate between *non-breaking*, *breaking and resolvable* and *breaking and unresolvable* changes. Non-breaking changes do not break instance models, whereas breaking changes can either be automatically resolved (resolvable) or require manual intervention (unresolvable). However, this classification is inadequate for our approach, since the potential to break instance models does not only depend on the change itself, but also on the immediate state of the example models. For instance, a global add node operation is a change that always makes the type of the added node mandatory. Performing the same change locally on only one example model can, but does not inevitably make a type mandatory. Figure 3.4 illustrates this: two example models define the connection of a node representing a personal computer (PC)

with a router and an access point (AP). Since the PC type is present in both example models, it is mandatory. When now a local add node operation with type “Router” is performed on the second example model, the type becomes mandatory as well. When the node is deleted again in only one example model, the mandatory constraint is removed again.



(a) Two minimal example models

(b) After local add of node 3 with type ”Router”, the type becomes mandatory

Figure 3.4: Example of a local add node operation that can break conformance.

Furthermore, it should be noted that the invalidation of an instance model by an example model change also depends on the instance model itself: If, in the previous example, an instance model already has a node typed by “Router”, it is not invalidated when the type becomes mandatory. Therefore, a universal classification of evolution scenarios without considering the actual state of the models is not helpful.

3.3.2 Classification of Scenarios

As pointed out in 3.3.1, every change at least carries the potential to break instance models, disregarding the scope of the change. Additionally, the question whether a change invalidates an instance model depends on the state of the example models as well as the instance model itself. Therefore, we do not classify the change operations by their potential to break instance models, but by their effects on the conformance relationship definition. Table 3.1 shows this classification. For every example model operation, its effect and the affected part of the conformance relationship as defined in 3.2.2 are shown. For instance, deleting a node can make the type of the node unavailable, potentially leaving nodes in the instance models untyped. Deleting a whole example model potentially affects every part of the conformance relationship, as it involves deleting all nodes, edges and attributes. In fact, an actual implementation could iteratively use the three operations “delete node”, “delete edge” and “delete attribute” to delete a complete model.

Change operation	Potential effect	Affected constraint
Add node	Make type mandatory	Type cardinality
Delete node	Makes type invalid	Node typing
Retype node	Makes type invalid	Node typing
Add edge	Makes edge mandatory	Edge
Delete edge	Makes edge invalid	Edge
Add attribute	Makes attribute mandatory	Attribute
Delete attribute	Makes attribute invalid	Attribute
Change attribute	Makes attribute invalid	Attribute
Delete model	Makes type invalid	Node typing
	Makes edge invalid	Edge
	Makes attribute invalid	Attribute

Table 3.1: Classification of example model changes

Of particular significance are changes that normally would be considered unresolvable when performed on meta-level, such as adding obligatory elements or restricting metaproperties [21]. In our approach, these correspond to globally adding nodes, edges and attributes, since they make the element obligatory and therefore need to occur in every instance model. One possibility to handle these scenarios is to classify them as unresolvable as well and ask the user to manually repair the conformance relationship by adding the obligatory elements to each instance model. However, the user has already given hints on how new elements should be included in the model by adding them to the example models. Therefore, it is possible to apply the same operation pattern on the instance models as well, making the changes automatically resolvable.

3.3.3 Resolving Issues

Once a change has been performed, the goal is to repair broken instance models. This is done in two steps: first, it is required to check if an instance model is broken. Second, if this is the case, the conformance relationship has to be repaired. Checking the validity of an instance model is possible by checking each of the conformance rules defined in 3.2.2. A naive method could first perform the change and then verify the complete conformance relationship. However, the classification also gives hints about which part of the conformance relationship is affected by which change. Therefore, it is possible and more efficient to only check the affected constraint for each instance model after a change. For instance, if an attribute has been deleted in an example model, it is sufficient to check if this change completely removes the attribute from the set of example models. If this is the case, the attribute becomes invalid. Therefore, every instance model with this attribute is broken.

In the next step, broken instance models need to be repaired. One method for doing this is to

inform users about the broken instance model and require them to resolve the issue manually by performing the necessary operations on the affected instance models. This treats all changes as unresolvable. However, to lower the cognitive load and aid the user, this process should be automated as much as possible. For every example model change, there is an equivalent instance model operation defined in 3.2.4. Together with the property that instance models do not conceptually differ from example models (and always conform to the same metamodel), broken instance models can be repaired fully automated by applying the same transformation that is used to alter the example model to the broken instance model. Table 3.2 extends the classification of example model changes by listing how a broken instance model can be repaired by applying the same operation to it.

Change operation	**Potential effect**	**Resolve issue by**
Add node	Make type mandatory	Add node with mandatory type (if not exist yet)
Delete node	Makes type invalid	Delete nodes with invalid type
Retype node	Makes type invalid	Retype nodes with invalid type
Add edge	Makes edge mandatory	Add edge between nodes
Delete edge	Makes edge invalid	Delete edge between nodes
Add attribute	Makes attribute mandatory	Add attribute to node
Delete attribute	Makes attribute invalid	Delete attribute from node
Change attribute	Makes attribute invalid	Change attribute of node
Delete model	Makes type invalid Makes edge invalid Makes attribute invalid	Delete node Delete edge Delete attribute

Table 3.2: Resolving broken instance models by using equivalent changes

Using this straight-forward scheme, it is possible to resolve all occurring issues automatically. Therefore, all instance models can be kept valid when the language changes. Figure 3.5 shows this concept based on the introductory example of a changing example model: by applying the same transformation T that was used to change the example model, the conformance relationship $R*$ between the instance- and the example model can be repaired. Since issues are resolved automatically and instance models modified without explicit user interaction, an actual implementation could point out the consequences of an example model change prior to applying it to raise the user's awareness of co-evolution taking place.

3.4 Concrete Syntax Modeling

In example-driven DSML design, the concrete syntax is usually defined together with the example models. Most already existing approaches presented in chapter 2 provide some kind of

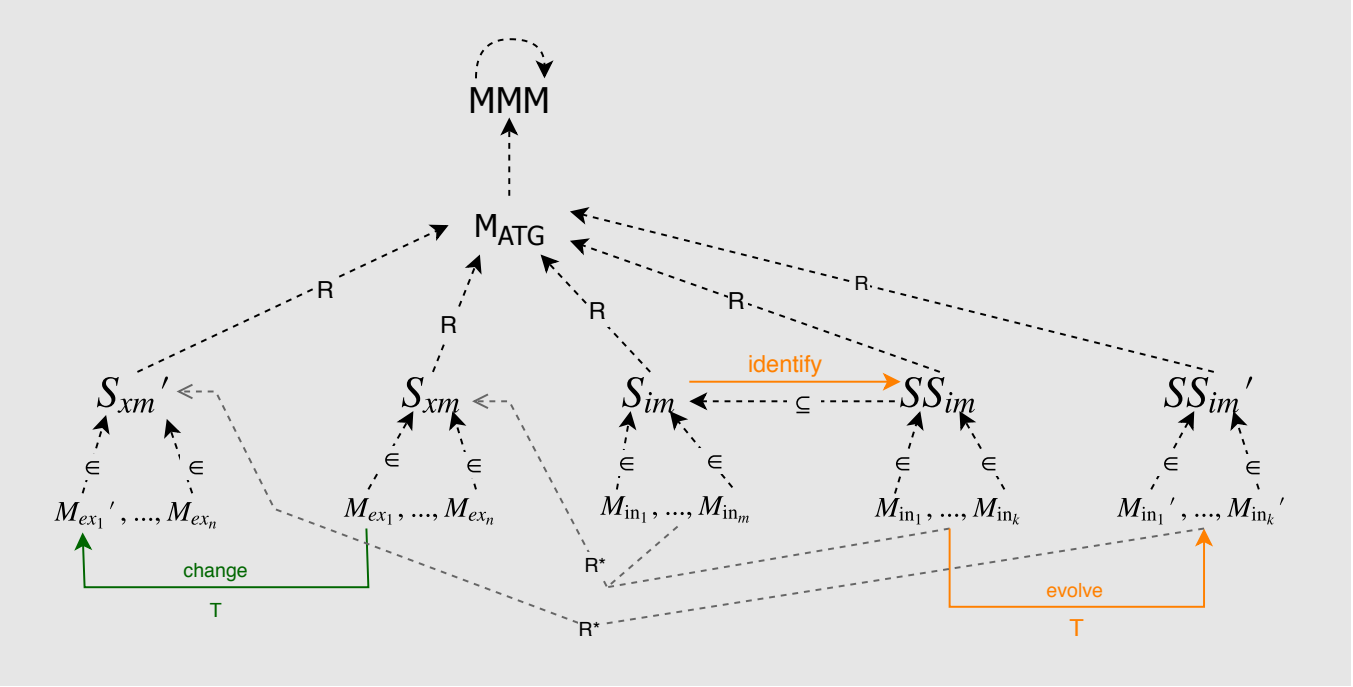


Figure 3.5: The co-evolution problem can be solved by applying the same change to the instance model

visual sketching interface where first, the concrete syntax of elements is defined. After that, these elements are enriched with type information, thereby establishing a link between abstract and concrete syntax. In our approach, we have so far only considered the abstract syntax of the language and deliberately refrained from imposing restrictions on how exactly example models are defined. However, one of the main goals of example-driven DSML design is to make use of the informal diagram sketching phase that typically happens during early system design and brainstorming and take such sketches as prescription of the abstract, but also concrete syntax of a modeling language. Therefore, a complete solution for DSML design must also take the concrete syntax into account, as formulated in requirement R2. This chapter explains the role of the concrete syntax in Moodling and argues that, just like the structure of the language itself, must be considered in the language evolution process. Section 3.4.1 first reflects on the role of visual concrete syntaxes in domain-specific languages in general. Then, section 3.4.2 describes how concrete syntax is embedded in our Moodling approach and how concrete syntax evolution is supported.

3.4.1 Visual Concrete Syntaxes

Visual languages consist of graphical *symbols* that together form the *vocabulary* of the visual language. Symbols from the vocabulary can be combined to *sentences* that form diagrams [87]. In modeling language design, visual notations are often neglected, especially when compared to the effort that is spent on defining the abstract syntax and semantics of a modeling language.

This is largely due to the lack of methods to measure the cognitive effectiveness and quality of visual notations with respect to a specific domain [85]. Hereby, the cognitive effectiveness defines how fast, accurate and easily a user can extract information from a visual symbol [62]. Often, language designers (that rarely have experience in graphic design) use notations that resemble established notations without questioning their cognitive effectiveness, leading to subpar results.

To overcome this issue, Moody has distilled a set of principles that help to design cognitively effective visual notations [86]. One of these principles is the *semantic transparency*: symbols should not only be different from each other, but also be visually linked to their meaning (for example a stickman to denote an actor in a UML use case diagram). A symbol which fulfills this requirement is called *semantically immediate*. If the symbol however has no relation to its meaning, it is called *semantically opaque*. Lastly, a *semantically perverse* symbol is a symbol that is adverse to its meaning, that is, a person that is not used to the notation would infer a different meaning.
Another principle is the *perceptual discriminability*, which is defined as how easy and accurate a user can discriminate between different symbols. This is largely influenced by the visual distance between the symbols. The greater this distance is, the faster and more accurately they can be recognized [119]. Moody argues that the shape should be the primary basis for discriminating between symbols, as the shape of an object is the main variable by which humans identify objects in the real world [86]. On the other side, he deems a differentiation between symbols of different types based solely on text and its typographic characteristics such as bold or italic as inefficient.

3.4.2 Concrete Syntax in Moodling

Overview

In contrast to traditional metamodeling tools, where the concrete syntax must be defined explicitly through e.g. icon languages [109], the concrete syntax in example-driven DSML design is usually given implicitly through the visual representations of the objects in the example models. In existing tools, two different methods have been used to define these example models: they are either sketched on some kind of digital canvas and processed by sketch recognition algorithms to identify individual symbols or the tool provides means to read image files generated with external drawing programs. Free-hand drawing tends to be faster since it is closer to the traditional pen-and-paper or whiteboard sketching, which is widely used to express ideas and discuss designs [35]. However, free-hand sketching and the accompanying sketch recognition algorithms almost always limit the user to the use of basic shapes with little visual detail. As a consequence, both the perceptual discriminability as well as the semantic transparency is likely to be reduced significantly. These problems can solved by using a general-purpose diagram editor such as *Visio* or *yEd*. They typically come with a rich set of predefined symbols for various

domains and allow for more detailed, pixel-precise drawing, which in turn makes it possible to design symbols with high perceptual discriminability and semantic transparency. However, using external visual editors introduces a disruption in the language design workflow as the user has to work with multiple tools. Furthermore, it requires the modeling environment to import and store the visual symbols used by the editor and provide a method to render them to screen. Lastly, it hinders the evolution of the concrete syntax from the modeling environment as the concrete syntax can only be manipulated by the diagram editor. When multiple example models are present, concrete syntax changes to a symbol must typically be executed manually on every example model. While in initial language design stages, the speed benefit resulting from free-hand sketching is of great advantage, we believe that the concrete syntax must be able to evolve together with the language's structure without switching between different tools. A common scenario that supports this argument is a user quickly sketching some symbols and annotating them with type information. Only later, when the language has evolved to a usable state, the user wants to replace the initial sketches with symbols that are cognitively more effective. Thus, the concrete syntax must be adaptable for new and changing requirements in a similar way the abstract syntax is.

Concrete Syntax Models

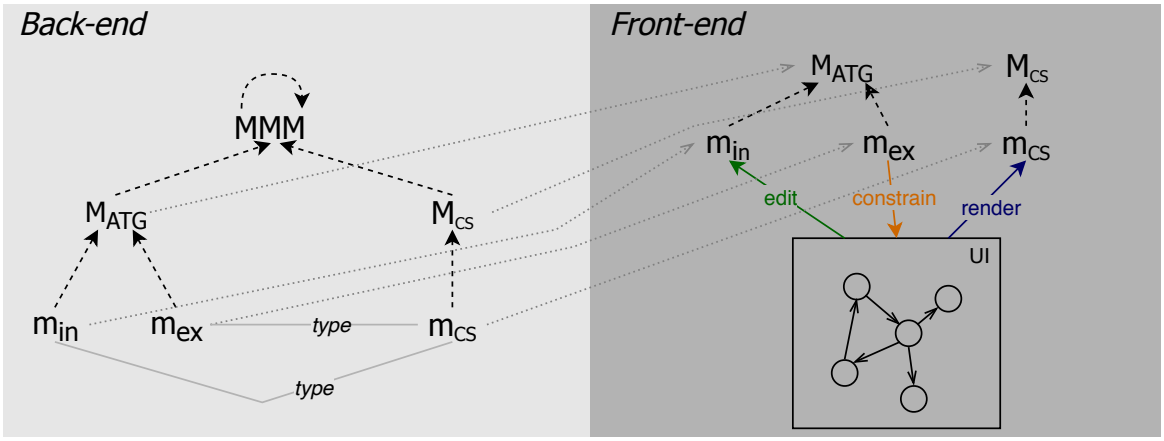


Figure 3.6: Overview of concrete syntax models in Moodling

Figure 3.6 gives an overview of an approach that enables concrete syntax evolution: the back-end, which must be a modeling environment with a notion of metamodels and conformance, stores both the instance- and example models as models of the ATG metamodel M_{ATG}. Additionally, it stores concrete syntax models, which conform to a concrete syntax metamodel M_{CS}. Therefore, all models reside in the same environment. The front-end retrieves these models from the back-end and uses the concrete syntax model to render an ATG model. A concrete syntax model holds information about the visual representation of a type of the ATG, i.e. how a type is rendered on the user interface. Such information can be stored in a plethora of ways,

for example an encoded image file or a collection of lines elements as a set of start and end positions. The user interface has to implement the corresponding methods to parse the model information and perform the necessary draw operations to render a node. Thus, the front-end must hold a copy of the concrete syntax metamodel M_{CS}. To establish a link between ATG models and concrete syntax models, the type information of a node is used as a unique identifier: for every type, there must be exactly one concrete syntax model. Therefore, a one-to-one relationship between a type and a concrete syntax model exists. For evolving concrete syntax models, the front-end implementation must provide a functionality to change the concrete syntax for a given type. A possible solution could be a concrete syntax editor, which is capable of retrieving, parsing, editing and storing the concrete syntax model. In case the concrete syntax model just describes image files, such an editor could be as simple as a small application which takes the image file and a type identifier and uploads a model constructed out of this data.

3.5 An Agile Moodling Process

In this section, we present an integrated and agile process to design domain-specific modeling languages. The process is agile in the sense that it is iterative and has a short feedback loop between design and usage. This allows the user to rapidly react to changing or new requirements. Furthermore, it is integrated as it considers all artifacts as models in the same environment that can evolve together as the language requirements change.

Figure 3.7 gives a high-level overview of the complete Moodling FTG+PM, which combines the activities and formalisms that were previously presented. Similar to the example-driven design process of existing tools (shown in figure 1.3), no initial requirements engineering activity has to be performed. Requirements are expressed directly by creating one or more example models. The example modeling activity yields example models, typed by an ATG, and concrete syntax models. Simultaneously, it also consumes the example models, since the types can be reused among different example models. Thereafter, instance models can be created by the instance modeling activity. This activity takes both the example models and concrete syntax models to provide a restricted environment for instance modeling, similar to how a metamodel would impose constraints on the instance modeling phase. During modeling, the engineer verifies the adequacy of the language. Again, since no formal requirements were defined, this verification step can be as simple as answering the question whether all aspects of the system to model can be expressed. If this is not the case, the language has to be revised by editing the example models and possibly the corresponding concrete syntax models. During this activity, all instance models are automatically evolved together with the changing example models. After the example models were changed accordingly, it is possible to immediately go back to instance modeling and see the results of the changes. Therefore, the feedback loop between language design and the actual usage is shorter when compared to approaches that explicitly generate a metamodel

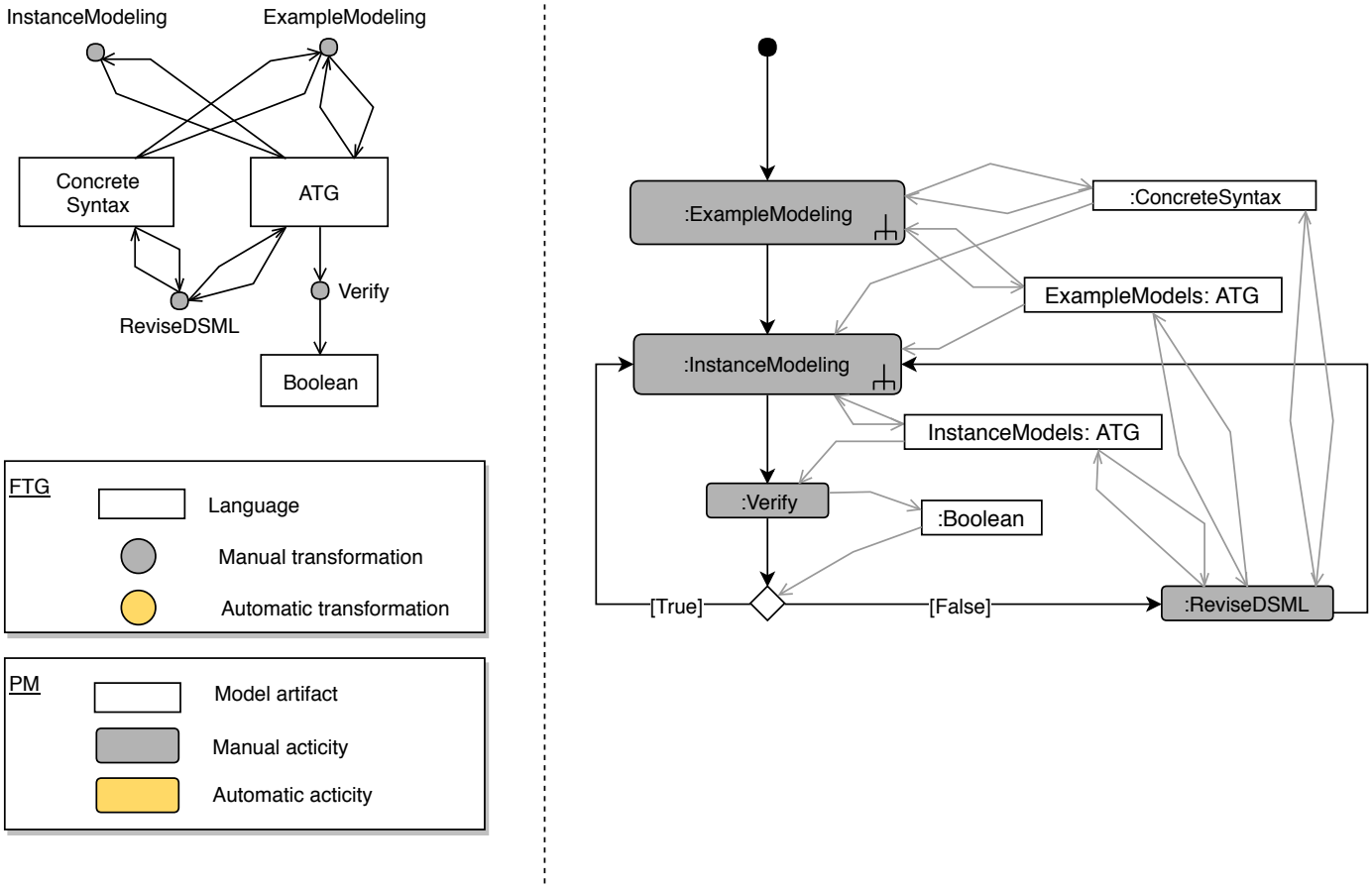


Figure 3.7: FTG+PM describing the top-level Moodling process

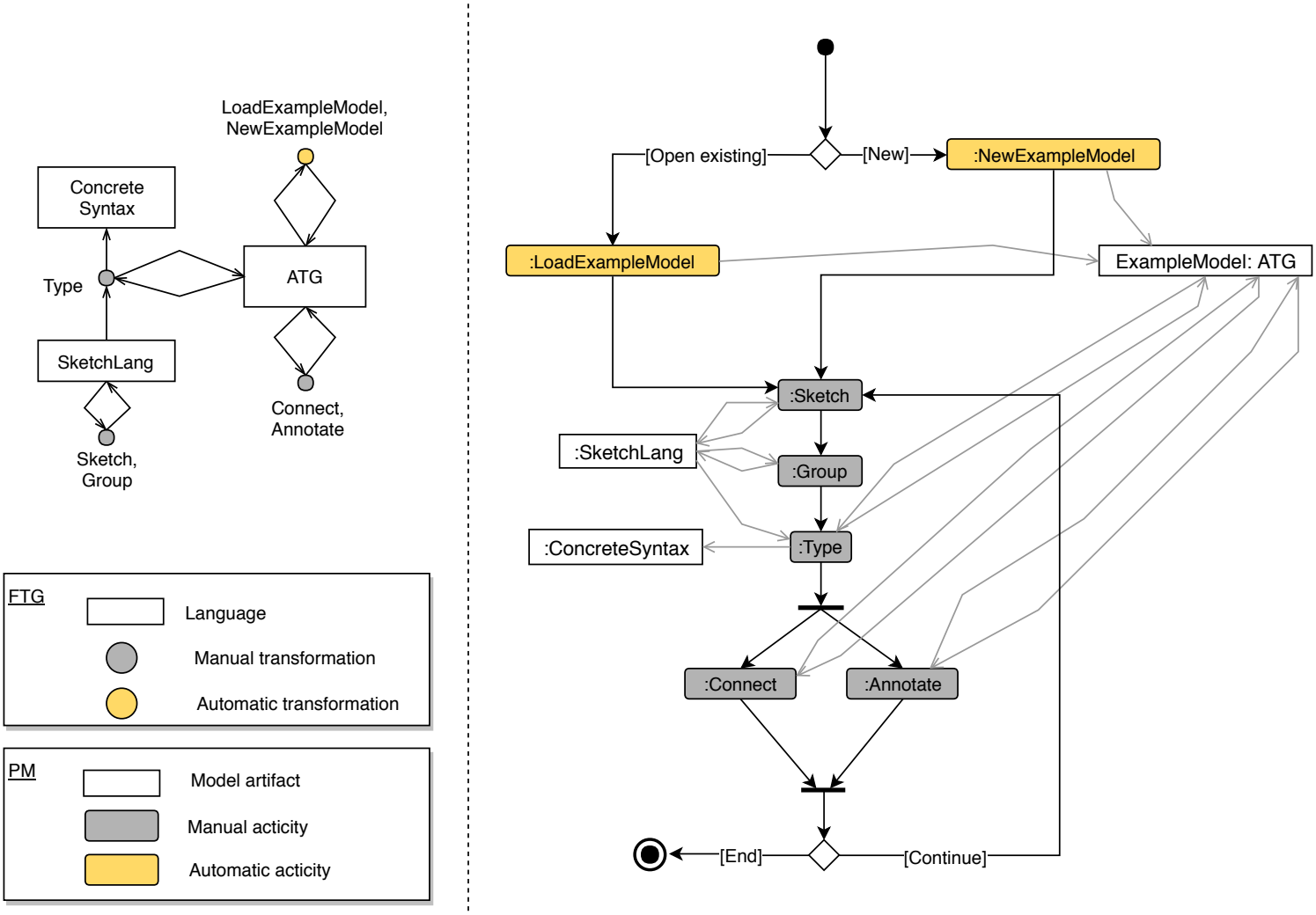


Figure 3.8: FTG+PM detailing on the example modeling activity

as an intermediate step. Therefore, we see requirement R4 as fulfilled. Since the example models are used to constrain the instance modeling phase as well as to evolve the language, requirements R1 and R2 are reflected in the process as well. In particular, since the example modeling phase does not only yield example models, but also concrete syntax models, which are manipulable as well, the concrete syntax can be evolved together with the abstract syntax. Furthermore, no metamodeling expertise beyond a basic understanding of the model structure and the constraints imposed by the example models is required to execute this process, which fulfills R5. Lastly, example models do not differ from instance models on a conceptual level, since both are typed by the same ATG metamodel. In fact, every example model can be used as an instance model by simply creating a copy and editing it during the instance modeling phase. Therefore, the process also reflects requirement R3.

Figure 3.8 details the example modeling activity. An example model can either be loaded (if it

exists) or a new one can be created. Both cases yield a model typed by the ATG metamodel. After that, a succession of operations are carried out repetitively to express language constructs in the example model: first, the sketching activity allows to create objects completely unconstrained, without the need to name or even structure them. Although the process does not specify how exactly this activity has to be implemented, it typically manifests itself as some sort of sketching functionality, where primitives can be drawn on a digital canvas. Therefore, the aforementioned objects can be visual primitives such as lines, rectangles and circles. The sketch activity yields a sketch, which is typed by a sketch language. After a set of objects have been sketched, they need to be structured to mark affinity between them. For instance, a set of primitives can represent one single element of the language. This is done during the grouping activity and is a necessary step to signify which objects belong to the same class. Grouped objects can be typed, that is, extended with information about the class they are instances of. The typing activity creates a new concrete syntax model by capturing the objects of the grouped sketch and persisting the information. By typing a group, a new node with the corresponding type information is created in the underlying model. After that, nodes can be connected or annotated. Connecting two nodes instantiates a new edge between them, while annotating creates a new attribute. Finally, the process can be repeated to iteratively build the example model. Note that during example modeling, neither the sketching nor the grouping activity are metamodeling activities and therefore do not modify the underlying ATG model. This can also be seen in the process model, since both activities do not produce the example model. On the other hand, typing, connecting and annotating objects are activities that directly modify the example model in-place.

Similarly, 3.9 shows an FTG+PM for the instance modeling activity of 3.7. Again, instance models can either be created from scratch or loaded if they exist already. Both activities produce an instance model that is typed by the ATG metamodel. Additionally, they also load the example models which have been previously defined during the example modeling activity. After an instance model has been loaded, multiple options are available. If the model is empty, instantiating a node is the only activity that can be performed. If it already contains nodes, it is also possible to instantiate edges and attribute nodes. All of these actions are executed in three steps: first, the user performs a manual add activity, which is followed by an automated verification step. It ensures that the corresponding operation is valid by querying the example models and searching for support for this operation. For nodes, this equals to searching for a node of the same type in any example model. For edges, any example model must contain an edge between the two types associated with the nodes to connect. Lastly, the attributing activity is verified by searching for an example model node with the same type and attribute. Only after the verification is complete, the corresponding element is instantiated in the instance model. Therefore, every activity in the instance modeling process is constrained. Furthermore and in contrast to the example modeling phase, all modeling activities in this process directly manipulate the ATG model.

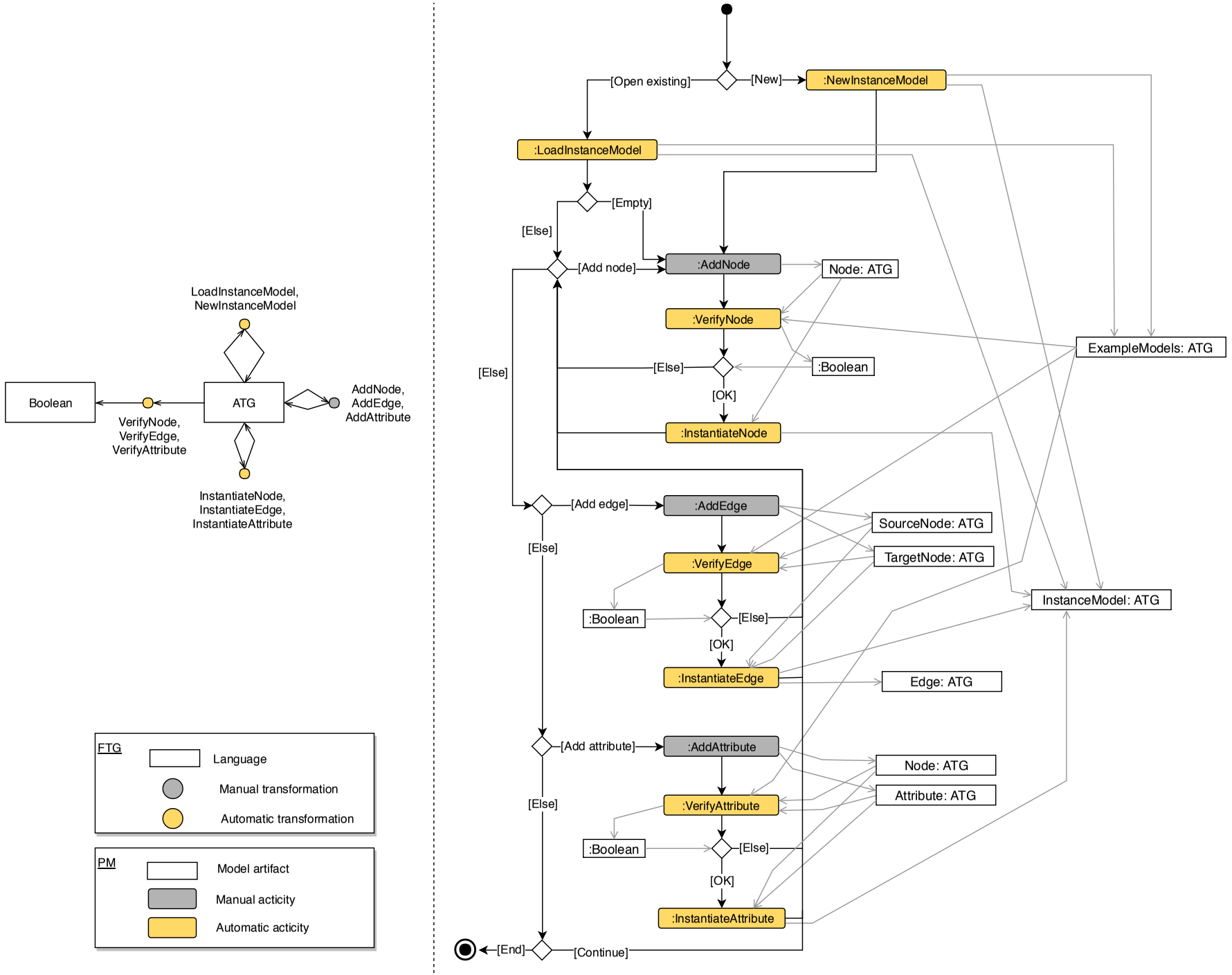


Figure 3.9: FTG+PM detailing on the instance modeling activity

Chapter 4

Implementation

This chapter presents a prototypical implementation of the theoretical concepts and the process that are described in chapter 3. First, section 4.1 presents the *Modelverse* as the underlying modeling framework for the implementation. Section 4.2 describes the ATG and concrete syntax metamodels. Section 4.3 elaborates on the concrete user interface which supports the Moodling process. In particular, it shows how sketches are subsequently transformed into example models and constrain instance modeling. Section 4.4 explains the implementation of the evolution transformations which ensure instance model conformance at every point in the process.

4.1 The Modelverse

The *Modelverse*[1] is a meta-modeling framework and model repository with support for Multi-Paradigm Modeling (MPM) [88][115]. In MPM, the goal is to "explicitly model all relevant aspects of the system using the most appropriate formalism(s), at the right level(s) of abstraction, while explicitly modelling the process"[114]. Therefore, the Modelverse supports language engineering through meta-modeling, model operations through model transformations and process modeling through process models. Hereby, every artifact is treated as a model itself. This includes metamodels, model operations and processes.

Model operations can be expressed by either using an imperative, explicitly modeled action language or declarative rules similar to the ones already presented in section 3.2.1. Model transformation rules are best used when it is easy to provide a mapping from constructs of the source language to constructs in the target language. In Moodling, retyping of a node could be expressed through a simple transformation rule which matches nodes of the old type and replaces them with nodes of the new type. On the other hand, using the action language is

[1] `https://msdl.uantwerpen.be/documentation/modelverse`

beneficial for operations where proven algorithms exist. For example, a reachability analysis of a Petri net can be much easier expressed using the imperative action language.

The Modelverse is accessible through an Application Programming Interface (API) which allows for CRUD operations, conformance checking and model and process execution [111]. Currently, a Python wrapper exists which wraps all these operations for developing applications that communicate with the Modelverse. For Moodling, we chose the Modelverse as underlying metamodeling framework since it provides a comprehensive environment for all modeling-related activities and does impose very little constraints on the developer. Furthermore, we chose Python as the implementation language since the only available API wrapper at the moment is written in Python.

4.2 Underlying Metamodels

With the simple class diagram formalism, the Modelverse provides a self-conforming meta-language for defining custom metamodels. For Moodling, two metamodels are required: one for the ATG and one for the concrete syntax models. Both are modeled with the class diagram formalism and explained in the subsequent sections.

4.2.1 ATG Metamodel

Figure 4.1 shows the metamodel for the ATG and therefore the structure of every instance and example model. Every model consists of a set of nodes, has a name identifier "name" and a boolean value "is_example" to signify the role of the model. Nodes can be connected by edges, have an identifier "ID" and a type identifier "typeID". Hereby, the "ID" corresponds to the label in the ATG and the "typeID" to the type of the node. Nodes can be connected to attributes with the *NodeAttribute* association. Contrary to the ATG definition, attributes are a tuple of key and value, whereby the key corresponds to the elements of the attribute vector from the ATG.

4.2.2 Concrete Syntax Metamodel

Figure 4.2 show the metamodel for the concrete syntax models. Every model stores the visual representation of a type from the ATG as an *Icon*. The "typeID" attribute of the Icon class links the concrete syntax model to its corresponding type. An additional boolean attribute "is_primitive" indicates whether the Icon is an actual image file or a group of geometric primitives. This shows the flexibility of the concrete syntax modeling approach, where an Icon can either be a group of sketched primitives or an image. An image file is stored in the class *Image* as a base64-encoded string attribute together with its scale factor. Groups of primitives are

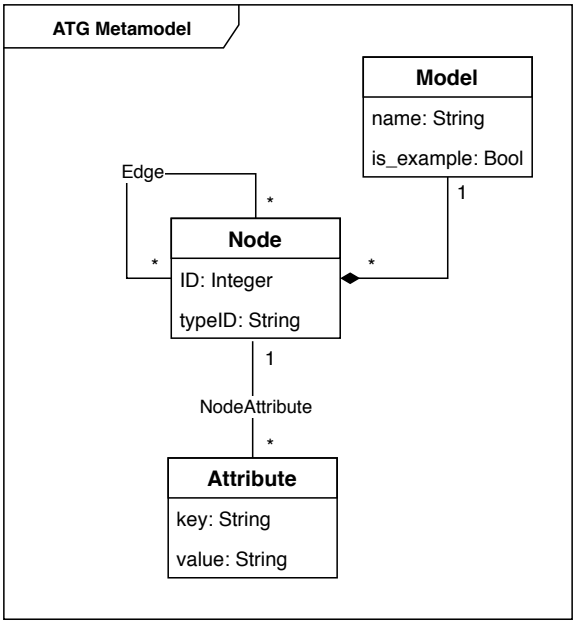


Figure 4.1: Metamodel for the attributed type graph

stored in the class *PrimitiveGroup*, which is associated with a set of *Primitive* classes. Hereby, a Primitive can either be a *Line*, *Rectangle* or an *Ellipse*. Every of these primitives store their position and shape size as attributes relative to the position of the Icon class itself.

4.3 The User Interface

The user interface (UI) of the implementation uses the *Qt* framework for displaying widgets and interacting with the user. The main factors that contributed to this choice are its maturity and wide-spread use, its extensive documentation, its rich set of predefined widgets and the availability of a Python wrapper. In particular, Qt provides a framework for displaying 2D graphical items, called the *Graphics View Framework*[2]. This framework forms the center of the UI, which consists of a canvas for displaying and manipulating models as well as sketching example model elements and a set of accompanying widgets to display attributes, available types and to give textual feedback. The reactive behavior of the UI is implemented using the Qt *Statemachine Framework*[3], which seamlessly integrates with the event system of Qt. The concepts and notations for the statemachines are based on Harel Statecharts [44] and the semantics on SCXML [6]. State changes are triggered by selecting different tools from the toolbar of the UI and enable or disable certain features such as dragging or selecting items.

The UI can be started in two different modes: Example modeling and instance modeling. Both

[2] `http://doc.qt.io/qt-5/graphicsview.html`
[3] `http://doc.qt.io/qt-5/statemachine-api.html`

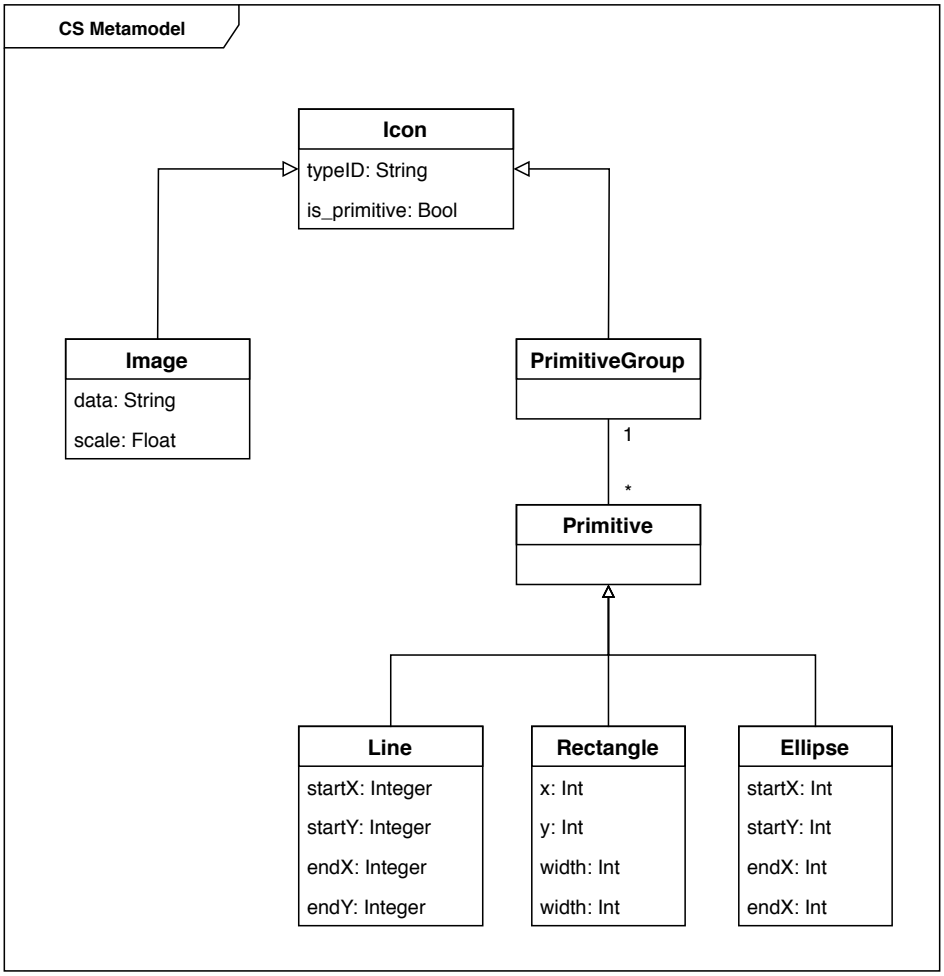


Figure 4.2: Metamodel for the concrete syntax

modes support the respective model manipulation operations shown in the theoretical concepts chapter 3. Additionally, an optional argument can be specified to open an existing model. If this argument is not present, a new, empty model is created. Figure 4.3 shows the top level behavior of the UI as a statemachine. While the example modeling state can be entered at any point in time, there must be at least one example model for the instance modeling state. By allowing the user to switch between the two modes, the iteration time becomes very short and new language constructs can be introduced at example model level at any point in time.

The following subsections elaborate on the two modes the user interface can operate in and give concrete examples based on the network DSML that has already been used in 2.

4.3.1 Example Modeling

In example modeling mode, the goal is to provide the user with a graphical interface to create and manipulate example models with minimal restrictions. Hereby, all example model operations

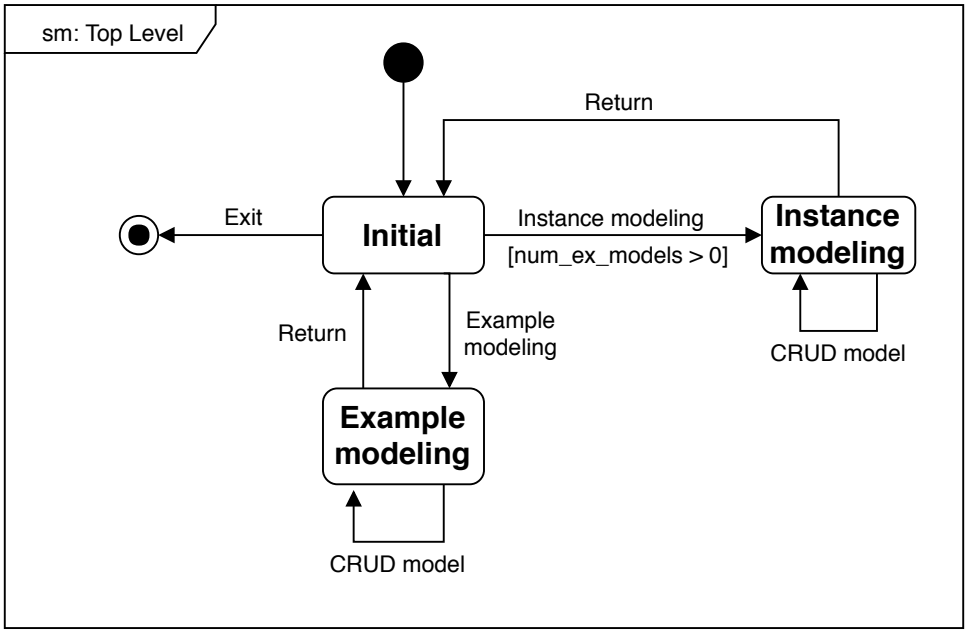


Figure 4.3: Statemachine describing the two UI modes

from 3.2.3 are implemented as model transformations in the Modelverse. As described in 3.3, many of operations can be either executed locally on the currently opened example model or globally on all example models and have the potential to invalidate instance models. Section 4.4 gives a more throughout explanation on how co-evolution is implemented. Concerning the UI, figure 4.4 shows a state chart that models the behavior of the example modeling mode. When an example model is opened, it is possible to freely sketch using the provided primitives. Primitives can be selected and grouped together. These groups can then be typed and annotated. Lastly, nodes can be connected via edges. The following paragraphs describe each of these states in more detail.

Sketching

Sketching example models is a vital aspect and one of the main reasons for example-based DSML design. Therefore, many already existing approaches try to mimic the pen-and-paper drawing process for their model creation phase. As shown in chapter 2, a plethora of different solutions were investigated for the sketching aspect, ranging from a simple drawing program to electric whiteboards with automated sketch recognition for hand-drawn shapes.
For our implementation, we provide a basic sketching canvas with support for graphical primitives such as lines, rectangles and ellipses. Most modern drawing applications offer a much richer set of graphical elements as well as the possibility to customize them with properties such as color and thickness. However, as already mentioned in 3.4.1, the shape of a symbol is the most fundamental visual variable for its perceptual discriminability. Therefore, we believe

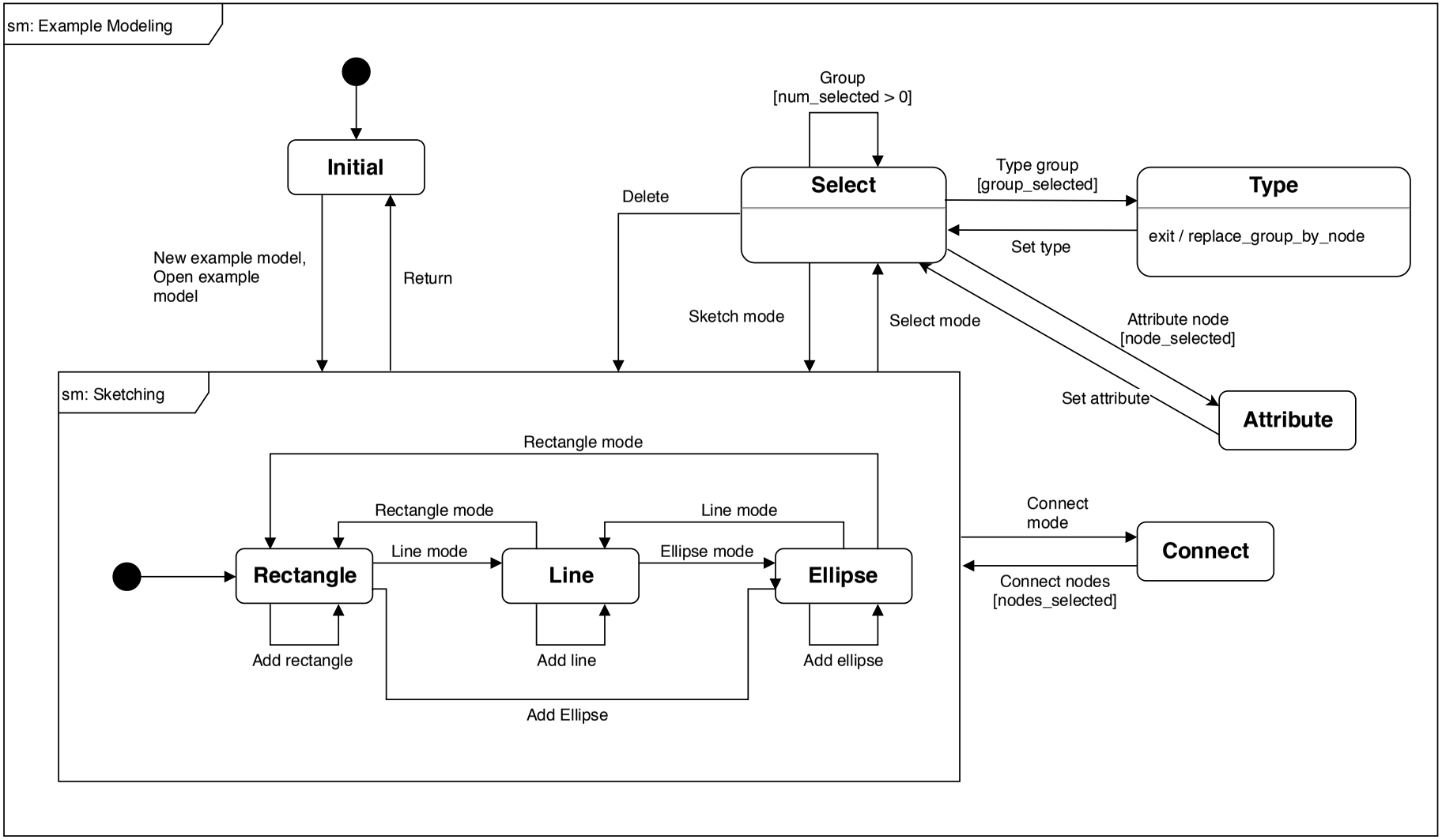


Figure 4.4: State machine describing the UI behavior when in example modeling mode

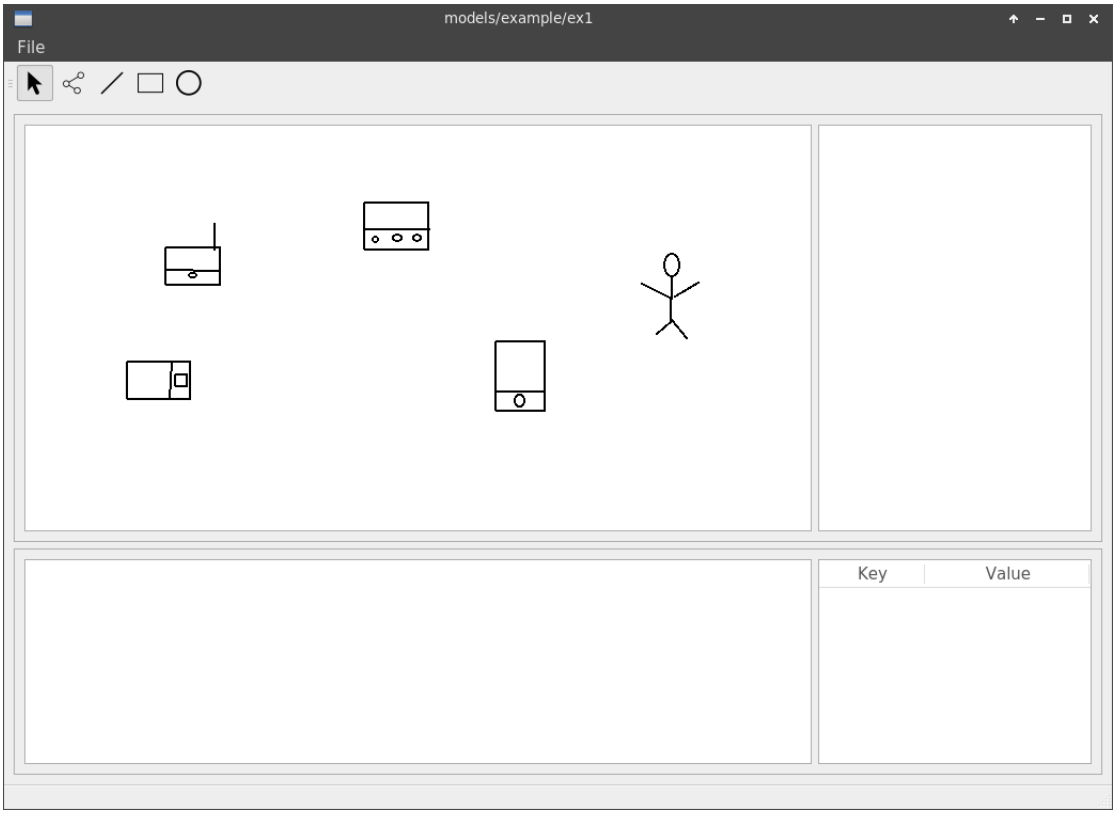


Figure 4.5: Screenshot of the tool in example modeling mode with sketches

that the basic primitives provided by our implementation suffice for sketching initial example models.

Figure 4.5 shows the example modeling UI with a set of primitives that eventually will form the nodes of the graph model. During sketching, the UI behaves like a general-purpose drawing tool, which means that no sketching activity modifies the underlying model. This enables the user to freely express ideas and concepts without being constrained by any meta-modeling aspects. Primitives can be selected, deleted and moved on the canvas.

Grouping and Typing

At one point, individual primitives need to be consolidated to groups. Again, this grouping is a purely graphical operation and does not modify the example model. One purpose of that activity is that grouped primitives behave like one unit. Therefore, moving and deleting a group affects all members of it simultaneously. Grouping primitives signalizes the user's awareness of type information and is a necessary prerequisite for subsequent modeling operations: once a set of primitives is grouped, it can be typed. By typing a group, the group is transformed to an actual ATG node. This activity includes the user to provide a type string, instantiating a new node in the underlying example model and capturing the concrete syntax of the group. Hereby,

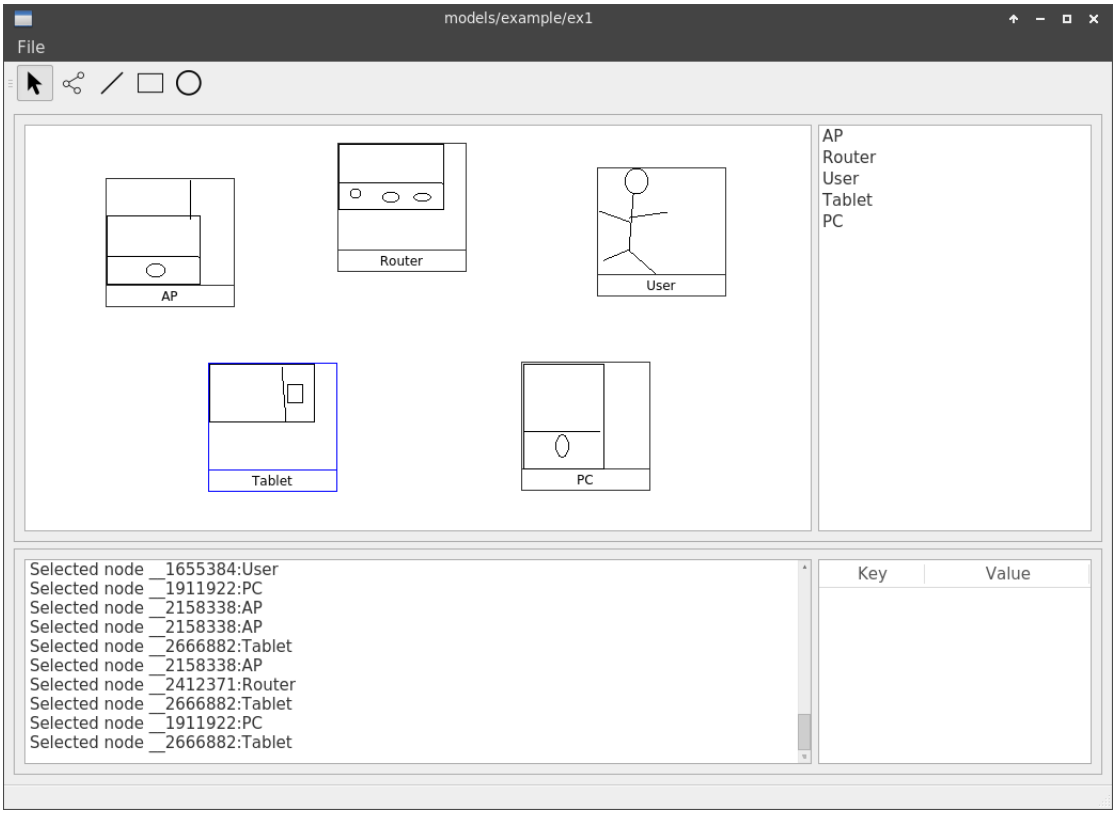


Figure 4.6: Screenshot of the tool in example modeling mode after typing

the type can either be a new type (that is, no other node in any example model already has this type), or an already existing type. In case the type already exists, the operation overwrites the concrete syntax of the type with the newly sketched group. This makes concrete syntax evolution for already existing types possible, as it effectively replaces the representation of the affected type in all models. Figure 4.6 shows the user interface after grouping and typing the sketches. Since this step adds a new type to the set of example models, the list of available types is updated accordingly. Furthermore, the sketches are replaced by the corresponding concrete syntax models which have been captured during the typing process and scaled to fit a fixed-width bounding rectangle that represents a node.

Edges and Attributes

Once groups have been typed and thereby been elevated to actual nodes in the model, they can be connected with edges and extended with attributes. Both operations directly manipulate the underlying model and are available as local or global change. Figure 4.7 shows the user interface after adding edges and attributes to the nodes.

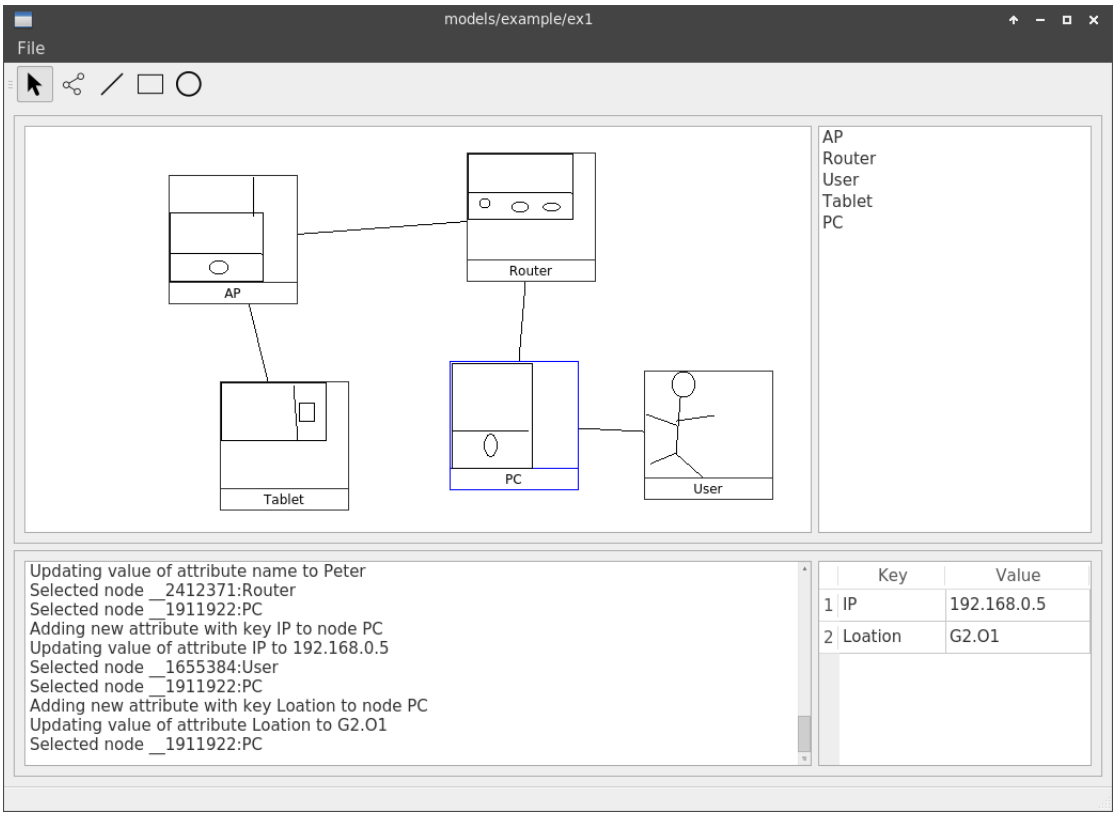


Figure 4.7: The tool in example modeling mode after connecting and attributing nodes

4.3.2 Instance Modeling

In instance modeling mode, the tool provides a constrained environment for creating instance models. Since all constraints are evaluated directly from the example models, the behavior of the UI directly depends on them. In this section, we first give a general overview of the UI in instance modeling mode and then elaborate on how the conformance relationship of 3.2.2 is implemented in our tool.

Overview

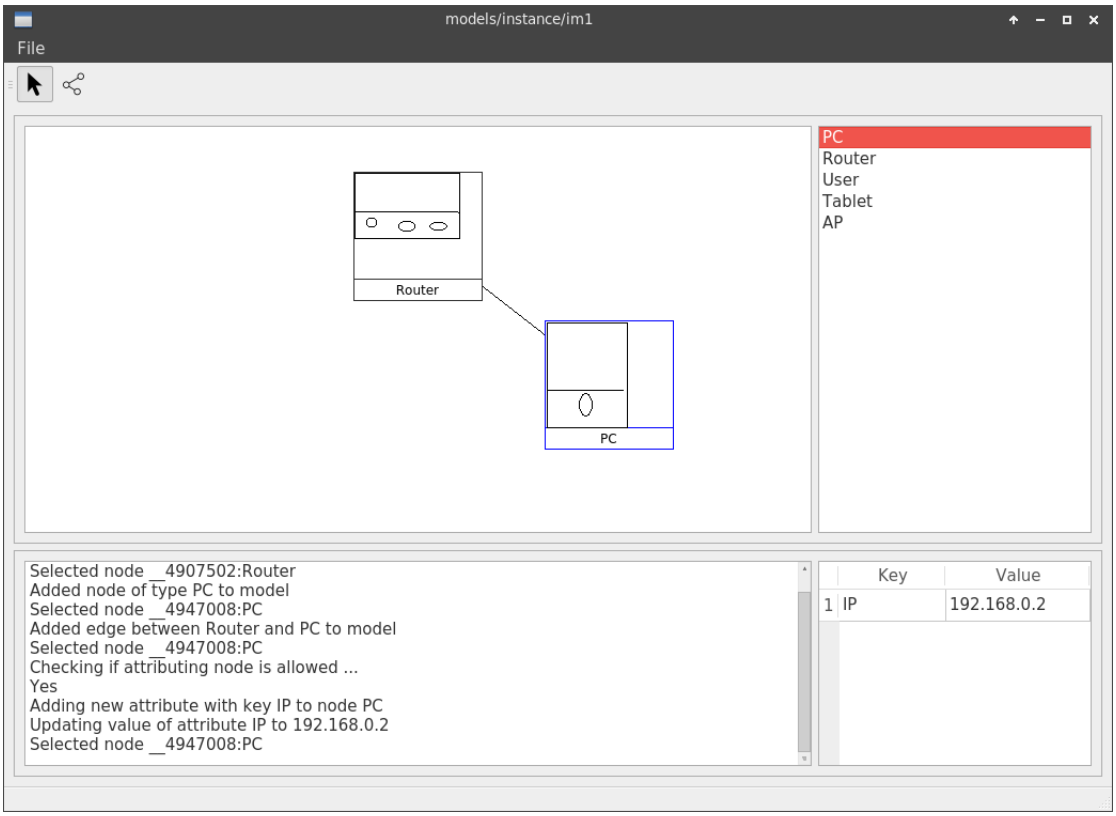


Figure 4.8: The tool in instance modeling mode

Figure 4.8 shows a screenshot of the UI with a loaded instance model. The list on the right is populated with the existing node types from all example models. By double-clicking on an item in this list, a new node of the respective type is instantiated in the model and its concrete syntax rendered on the canvas by fetching the concrete syntax model from the Modelverse and calling the appropriate Qt drawing functions such as *addLine* or *addRectangle*. The toolbar provides a set of manipulation methods for the user. During instance modeling, these methods are essentially selecting objects on the canvas and connecting them with edges. All other activities such as deleting objects or attributing nodes are triggered via keyboard shortcuts.

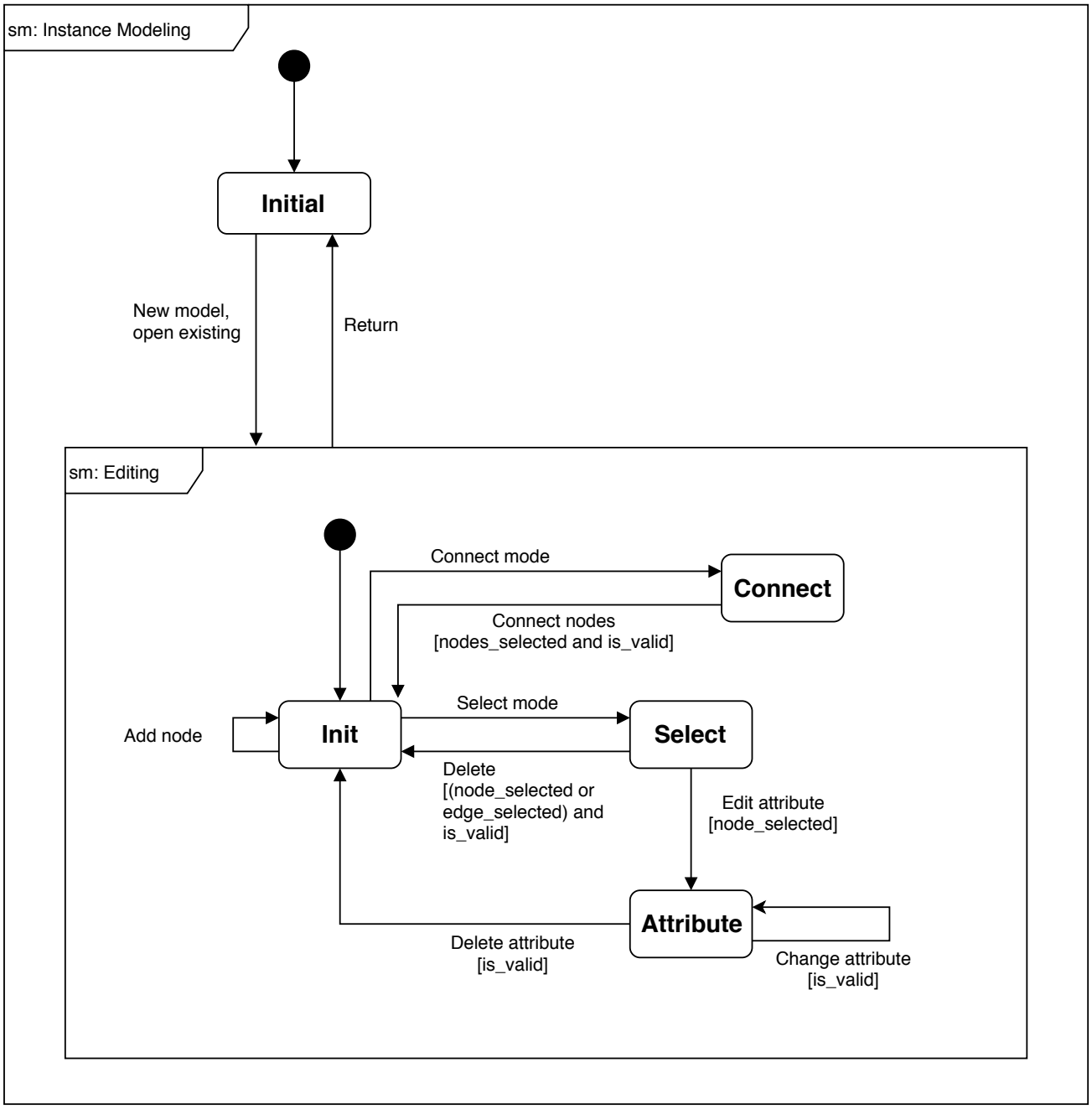


Figure 4.9: State machine describing the UI behavior when in instance modeling mode

Figure 4.9 depicts the UI behavior when in instance modeling mode as a statemachine. All operations described in 3.2.4 are supported with their respective constraints: in essence, nodes with the types from the example models can be instantiated, connected and attributed. This is reflected in the three states *Connect, Select* and *Attribute* in the statemachine. Furthermore, the constraints for certain operations are represented as transition guards. For instance, connecting nodes is a only valid if a connection between the same types exists in any example model. Because of the constraints, the UI behaves much more like a traditional meta-modeling environment. As a consequence, it is not possible to sketch on the canvas and introduce new types or previously non-existing edges or attributes.

Implementation of the Conformance Relationship

During instance modeling, all constraints are evaluated on-the-fly from the example models. For that, a wrapper around the Modelverse API was developed that provides a set of ATG-related methods. For instance, it implements a method to check if an edge is valid between two types by iterating through all example models and checking if such an any of them has an edge between the two types. If this is not the case, the operation is invalid and not executed. In 4.1, an implementation of the method *is_edge_supported* is shown. This method is called prior to adding an edge to an instance model, takes two types as parameters and returns a boolean that signifies if an edge between the two types is supported. For each example model, the method retrieves all nodes with the corresponding types and checks if there exists an association between them. If this is the case in any example model, the edge is valid.

```
1  def is_edge_supported(from_type, to_type):
2      for m in all_example_models():
3          nodes_from = all_nodes_with_type(m, from_type)
4          nodes_to = all_nodes_with_type(m, to_type)
5          for nf in nodes_from:
6              for nt in nodes_to:
7                  assocs = get_associations_between(m, nf, nt)
8                  if assocs:
9                      # somewhere in an example model, there is such an
                           association
10                     return True
11     return False
```

Listing 4.1: Method to check if an edge between two types is valid

The method uses two helper functions: *all_nodes_with_type*, to get all nodes with a specific type from an ATG model and *get_edges_between*, which returns all edges between two nodes. Their implementations are shown in 4.2 and 4.3, respectively. Both of them use methods provided by the Modelverse API to directly access the underlying models. Note that *get_edges_between*

reads all outgoing and incoming edges from both nodes and returns the intersection of them, since associations in the Modelverse Simple Class Diagram formalism are always directed.

```
def all_nodes_with_type(model, node_type):
    all_nodes = mv.all_instances(model, "Node")
    ret = [node for node in all_nodes if mv.read_attrs(model, node)["typeID"]
        == node_type]
    return ret
```

Listing 4.2: Helper method to return all nodes with a specific type in a model

```
def get_edges_between(model, node1, node2):
    edges_n1 = mv.read_outgoing(model, node1, "Edge")
    edges_n1.update(mv.read_incoming(model, node1, "Edge"))
    edges_n2 = mv.read_outgoing(model, node2, "Edge")
    edges_n2.update(mv.read_incoming(model, node2, "Edge"))
    return list(edges_n1.intersection(edges_n2))
```

Listing 4.3: Helper method to return all edges between two nodes

Figure 4.10 shows an example where the UI informs the user that an edge between two types is not valid since no example model defines an edge between the elements “Router” and “Tablet”.

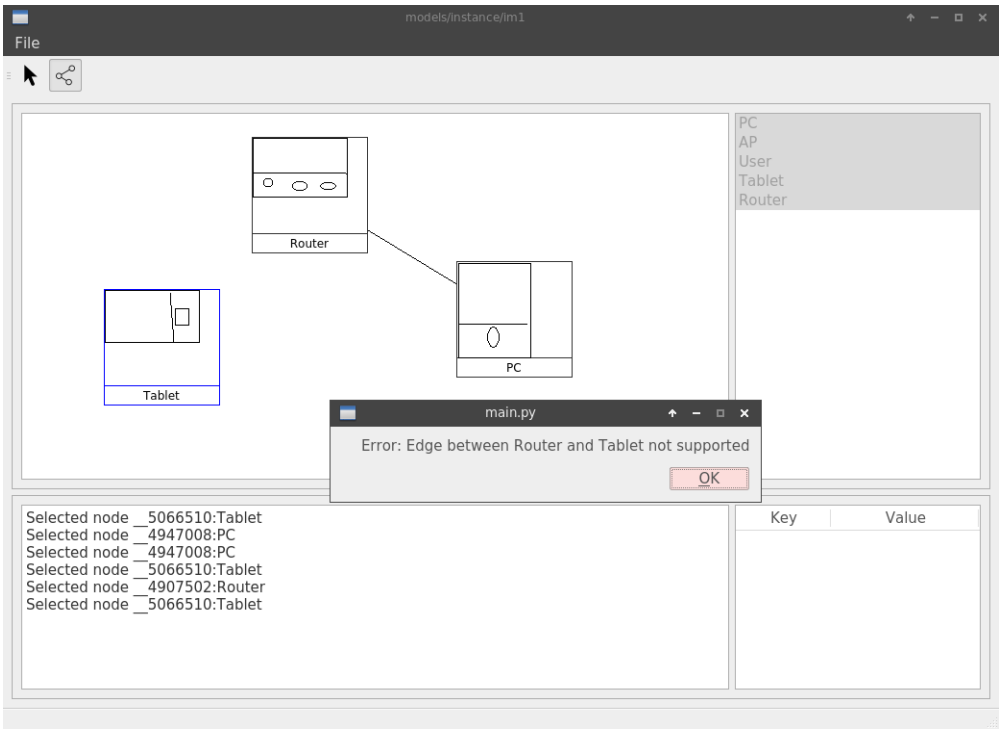


Figure 4.10: Example of the constrained instance modeling mode

However, constraining the modeling operations is not sufficient to ensure conformance, as the

following example illustrates: Consider a set of example models which together define a mandatory type τ_i. Then, any instance model which does not contain at least one node that is typed by τ_i is invalid. In our approach, it is possible to construct such an instance model by creating a new instance model, which will be initially empty. If a node with τ_i is never added by the user, the instance model remains invalid.
The same holds true for mandatory edges and attributes. Therefore, it is necessary to verify these constraints independent of any instance modeling operation (in fact, it is even impossible to do so). Multiple implementations of such a verification method are possible: the instance model could be verified continuously, before the model is closed (and the modeling phase exited) or manually upon user request. Since the verification of the instance model is a potentially costly operation, we decided against a continuous verification. Only validating the instance model once before the modeling phase is exited disallows frequent feedback for the user during modeling. Therefore, we implemented a verification upon user request, since it allows the user to verify the currently opened instance model at any point in time. It can be triggered by an option in the menu of the UI. The verification code itself is implemented as a dedicated class and accesses model properties by using the wrapper. Listing 4.4 shows the code of the mandatory type verification method. First, all mandatory types are computed by iterating through all available types defined by the example models and checking for the mandatory attribute. Then, for all mandatory types, it is checked if the instance model to verify contains at least one node of this type. If not, the verification failed and the instance model is invalid. Note that this verification has no consequences further than informing the user about the invalidity of the instance model. As a consequence, it is still possible to exit the modeling phase and store an invalid instance model in the environment. One strategy to counter this issue is to perform the verification automatically prior to exiting the phase.

```
1  def verify_mandatory_types(self):
2      # get list of all mandatory types
3      type_mandatory = {t:False for t in self._available_types}
4      for node_type,_ in type_mandatory.iteritems():
5          type_mandatory[node_type] = commons.is_type_mandatory(node_type)
6      mandatory_types = [node_type for node_type, mandatory in
           type_mandatory.iteritems() if mandatory]
7
8      # check if mandatory types are in instance model
9      for mtype in mandatory_types:
10         all_of_type = commons.all_nodes_with_type(self._instance_model, mtype)
11         if not all_of_type:
12             return {"OK": False, "error": "Mandatory type {} not
                   found".format(mtype)}
13
14     return {"OK":True, "error":None}
```

Listing 4.4: Code to verify mandatory type attribute for an instance model

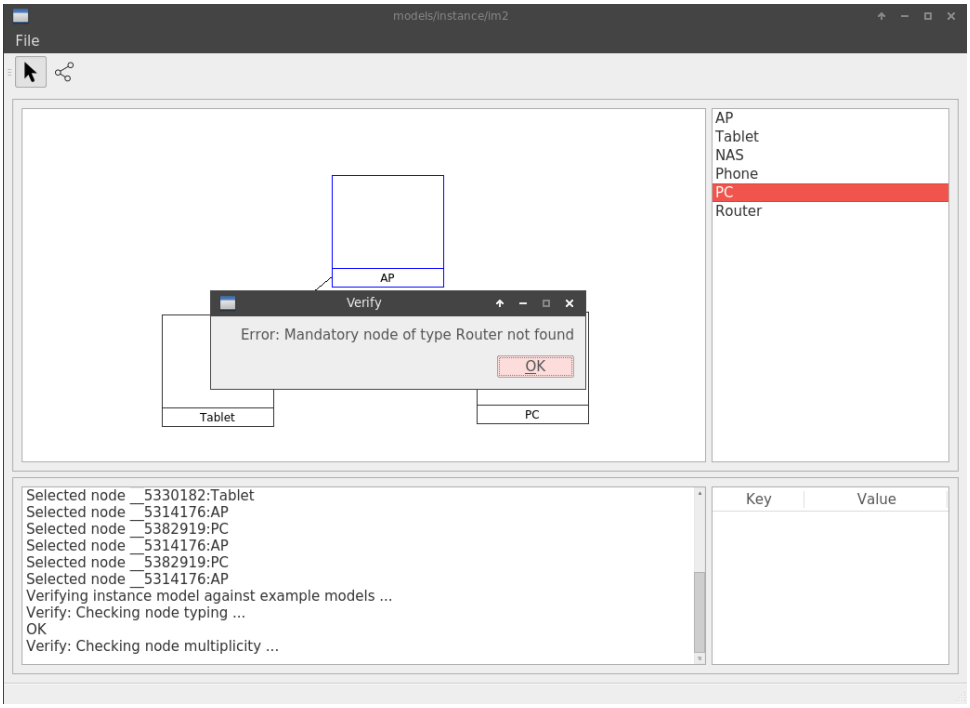


Figure 4.11: The UI informs the user about a failed verification

In the front-end, the user triggers the verification process manually by selecting it from a menu. While the algorithm runs, the UI is disabled and the result of the individual verification steps are printed on the screen. If the verification fails, the user is notified about the exact error. In figure 4.11, such a notification is shown for an instance model that does not contain a mandatory type.

4.4 Co-Evolution

When in example modeling mode, all model operations requires a successive evolution step to check and, if required, repair the conformance relationship between the changing example model and the instance models. In our tool, all example model operations are implemented using model transformations. Since all models conform to the same ATG metamodel, these operations can be executed on instance models as well. This property is used to automatically repair the conformance relationship if required: whenever an example model operation is executed, it is checked if it invalidated an instance model. Then, the same operation is executed on the instance model to repair the conformance relationship.

The Modelverse allows implementing model transformations as declarative graph transformation rules or with an imperative action language. In our implementation, we investigated both approaches and found them to be too static for our needs. Most transformations require one or more arguments that are determined only at run-time, often from user input (for example,

retyping a node requires the new type as parameter). While passing data to the action language is possible in theory, it introduces a significant overhead by requiring a communication model as statechart. Therefore, we decided to use a manual transformation which is provided by the Modelverse as a method to modify a model in-place using a callback function. This callback function defines the transformation as a series of Modelverse API calls and therefore can use all concepts of the underlying programming language.

Listing 4.5 shows the evolution code for adding a node to an example model. The class exposes two methods to the caller, namely "execute" and "repair". The execute method adds a node with a specified type to a model. If the scope of the operation is local, the corresponding manual model transformation is called with the model and the callback method as parameter. The callback function then instantiates a new node in the given model and assigns the type attribute to it. If the scope is global, the method iterates through all example models and calls itself recursively on each model with local scope. The repair method, which is typically called after the operation was executed on an example model, checks if the operation made the node type mandatory. If this is the case, a node with that type is added to all instance models which do not already have such a node. This is done by executing the model transformation again, but this time on the instance model.

```
1  from wrappers import modelverse as mv
2  import commons
3
4  class NodeAdd(object):
5      def __init__(self):
6          self._node_type = ""
7
8      def execute(self, model, node_type, local):
9          self._node_type = node_type
10         if local:
11             mv.transformation_execute_MANUAL("graph_ops/add_node",
12                     {"gm":model}, {"gm":model}, callback=self._callback)
13         else:
14             for m in commons.all_example_models():
15                 self.execute(m, node_type, True)
16
17     def _callback(self, model):
18         node_id = mv.instantiate(model, "gm/Node")
19         mv.attr_assign(model, node_id, "typeID", self._node_type)
20
21     def repair(self):
22         if commons.is_type_mandatory(self._node_type):
23             for im in commons.all_instance_models():
24                 if not commons.all_nodes_with_type(im, self._node_type):
25                     self.execute(im, self._node_type, local=True)
```

Listing 4.5: Evolution handler for adding a node

Line *22* checks if the operation made the type mandatory by calling "is_type_mandatory". This method is implemented in a dedicated module called "commons", which functions as a layer between the Modelverse API and the model manipulation code. Listing 4.6 gives the source code of this method. Essentially, it iterates through all the example models and retrieves a list of nodes with the respective type. If no such nodes exist in any example models, the type is not mandatory.

```
1 def is_type_mandatory(node_type):
2     for exm in all_example_models():
3         nodes_with_type = all_nodes_with_type(exm, node_type)
4         if not nodes_with_type:
5             return False
6     return True
```

Listing 4.6: Method to check if a type is mandatory

All scenarios that were identified in section 3.3.2 are implemented in a similar fashion, resulting in a total of eight classes for the eight different changing operations. An exception is the deletion of an entire example model, which is done by iteratively calling the delete handlers for nodes, edges and attributes on the example model. This ensures that instance models are evolved automatically in case any objects become unavailable due to the delete operation. As described in section 3.3, many example modeling operations can be executed locally (on the current model only) or globally (on all example models). The UI implements this by querying the user prior to executing an example modeling operation. Figure 4.12 shows such a query for deleting a node. After the user has selected the scope of the operation, the corresponding evolution handler is executed with the respective parameters.

Concrete syntax evolution is supported during the example modeling phase: when a group of sketched primitives is typed and the type equals an already existing type, the concrete syntax of the existing type is overwritten with the new sketch. Furthermore, we have implemented a command-line tool to upload an image file as concrete syntax model. This makes it possible to replace sketched symbols by visual notations with a richer set of variables than our sketching interface currently allows. An example where all sketched symbols were replaced by icons can be seen in 4.13.

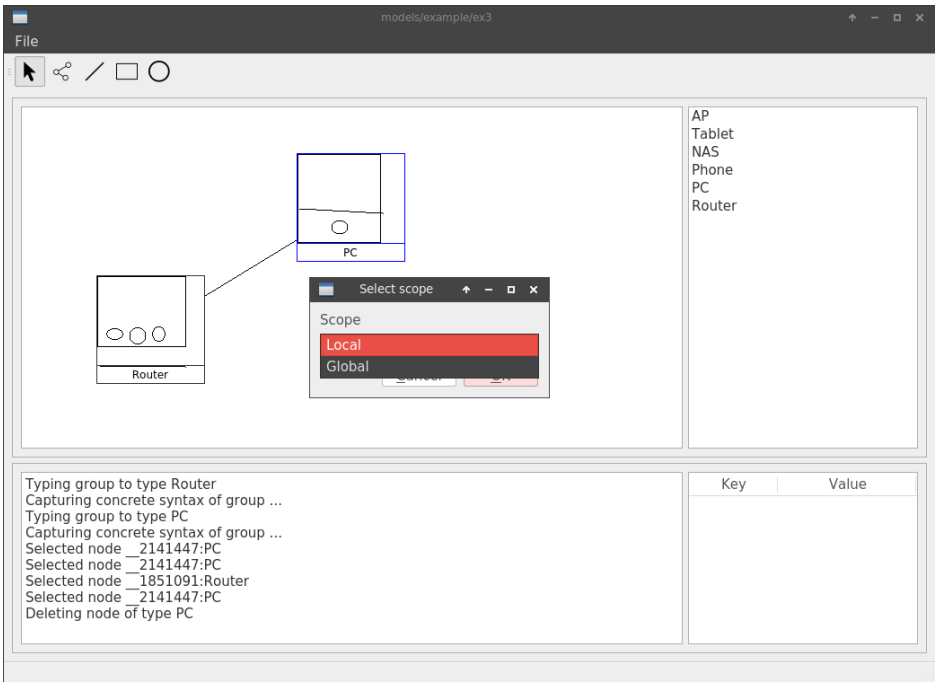


Figure 4.12: Screenshot of the scope selection before performing a delete operation on a node

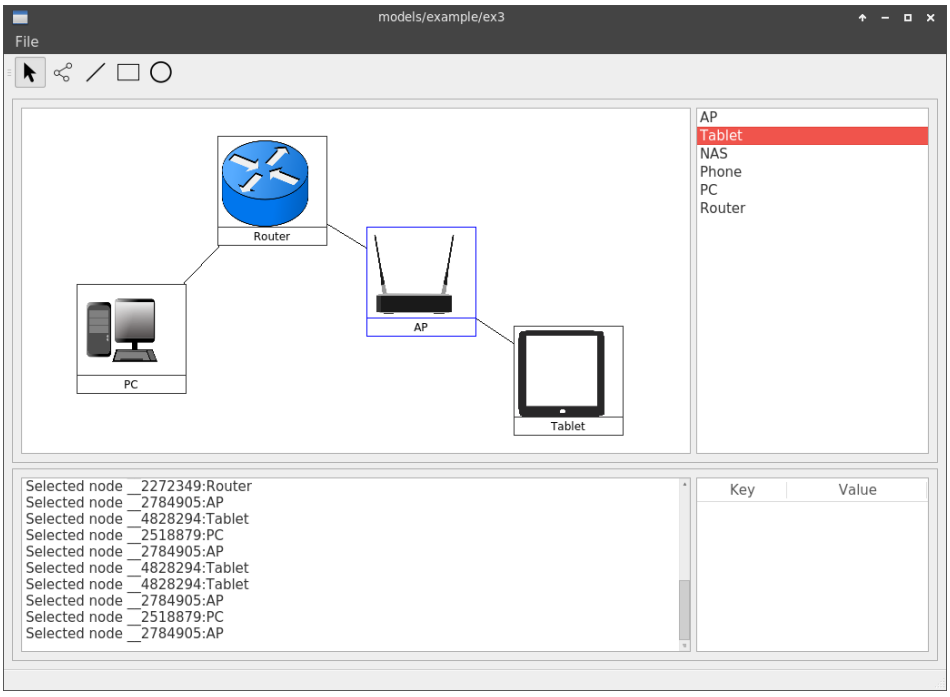


Figure 4.13: Concrete syntax evolution: sketched primitives are replaced by icons

4.5 Process Modeling

In 3.5, we presented a process that chains all the activities in the approach together. Since the Modelverse supports process modeling through the FTG+PM formalism, we attempted to explicitly model the process using the provided formalism. However, the Modelverse requires fixed strings to describe the location and name of a model artifact that is consumed or produced during an operation. An example of this is shown in 4.7, where the two activities to create and edit an example model are linked together. First, the transformation *new_exm* is executed that produces a new model artifact in the Modelverse. This model artifact is an example model located at the Modelverse path "models/example/exm1". Then, the transformation *edit_exm* consumes this model and modifies it in place by producing it as the same output again. Realistically, this transformation would start the UI with the consumed model as parameter to allow the user to modify it.

```
Start start {}
Finish finish {}

Exec new_exm {
    name = "process/new_exm"
}
Exec edit_exm {
    name = "process/edit_exm"
}

Next(start, new_exm) {}
Next(new_exm, edit_exm) {}
Next(edit_exm, finish) {}

Data exm {
    name = "models/example/exm1"
    type = "formalisms/graphMM"
}

Produces(new_exm, exm) {
    name = "new_example_model"
}
Consumes(edit_exm, exm) {
    name = "example_model"
}
Produces(edit_exm, exm) {
    name = "example_model"
}
```

Listing 4.7: Excerpt from the modeled process to create and edit an example model

Since our approach supports an unlimited amount of example models, it is impossible to anticipate the amount of data nodes in the process model. This is not only limited to example models, but also instance and concrete syntax models. Therefore, the current FTG+PM formalism as implemented in the Modelverse is well suited for processes with a fixed set of model artifacts, all of which are known beforehand. For our approach however, the user currently has to resort to executing our tool with the respective parameters by hand. Nevertheless, we give an executable sample process with hard-coded model artifact paths in the appendix B. This process also models the instance modeling and concrete syntax activities.

Chapter 5

Evaluation

This chapter evaluates the presented approach and the accompanying prototypical implementation. First, section 5.1 defines a set of research questions, based on the goals and requirements set in 3.1. Section 5.2 discusses these questions individually and reflects upon whether the presented approach meets the initial goals. Section 5.3 extends the scope of the evaluation by linking back to chapter 2 and comparing our approach with other solutions.

5.1 Research Questions

In this work, we aimed to design and implement an agile and integrated process for example-driven DSML creation. To evaluate our approach and verify that it meets the requirements we formulated in 3.1, the following research questions are distilled:

1. **R1:** Is it possible to react quickly to new and changing language requirements?
2. **R2:** Can the approach be used without metamodeling experience?
3. **R3:** Is the process integrated in a single environment?

R1 is concerned with the agility of the approach. One goal of Moodling is to implement an agile process that makes it possible to react to changing requirements by keeping iterations short and building the DSML iteratively. This poses a particular challenge with regard to the co-evolution problem, where instance models break when the definition of the DSML changes. R2 reflects on how suitable the approach is for users without meta-modeling experience. This is an important question since example-driven DSML design aims to make domain-specific languages more accessible for domain engineers that do not have language engineering experience. This also includes the cognitive load of the approach, which is dependent on the number of model

artifacts exposed to the user. Finally, R3 aims at the integration of the process. All model artifacts must reside in the same environment and be accessible throughout the whole process. This implies that all models can be subject of model transformations that allow for analysis, execution and code generation.

5.2 Results

R1: Agility We have implemented an agile, example-driven DSML design process that makes short development iterations possible by omitting the metamodel generation step and inferring all language constraints directly from the example models. The process is divided into two different activities, namely example modeling and instance modeling. During example modeling, models that serve as examples of valid language models can be created in an unconstrained manner. During instance modeling, the previously generated example models are used to constrain and impose restrictions to the modeling process. Furthermore, we have shown that fully automated co-evolution is feasible with our approach, since all models conform to the same metamodel and can evolve together by applying the same model transformations that change the example models to the instance models. Lastly, our approach does not only consider the abstract syntax, but also the concrete syntax of the DSML, which is stored as a set of concrete syntax models and can evolve as well. As such, it is possible to react to changing requirements by integrating them in the example models and letting the instance models evolve automatically.

R2: Required experience We do not expose a metamodel to the user and infer the abstract syntax of the DSML directly from the example models. Therefore, users always work on the level of concrete objects and not on meta-level. This limits the amount of manipulable artifacts and is more natural for non-experts. Furthermore, we make a clear distinction between the language design and modeling phases, which further helps to reduce the cognitive load of our approach. Switching between these phases can happen at any point in the process, since all instance models evolve automatically without requiring manual user intervention. However, a basic knowledge of modeling in general and connection-based languages in particular is required to use the approach, since all models are instances of a graph-like data structure. Furthermore, to effectively use the tool during the example modeling phase, users must be aware of the implications of their actions with regard to the constraints that are imposed on the succeeding instance modeling phase. For instance, a mandatory type constraint can be modeled by including a node of this type in every existing example model. Lastly, the implemented tool includes elements to improve usability: When an instance model is verified, non-conforming parts are highlighted and a warning about the violated constraint(s) is issued. Furthermore, textual feedback is given during example modeling mode when an operation changed a constraint. For instance, typing a node can result in creating the mandatory type constraint. However, creating example models generally consists of a finite iteration of the activities sketch, group, type

and annotate. Although these activities are represented in the user interface, their order is not evident without knowledge about how the tool works. Lastly, a more throughout feedback about the consequences of an example model operation, especially with regard to co-evolution is required to increase the usability and the users' awareness of their actions.

R3: Integration Our approach is designed for and implemented in a single metamodeling environment, the Modelverse. Example models, instance models and concrete syntax models are all stored at the same location. Since the Modelverse is a multi-paradigm metamodeling environment, all common metamodeling activities such as code generation or model analysis can be implemented and performed without needing to export any model beforehand. Furthermore, example models do not differ from instance models, since they conform to the same ATG metamodel. Thus, example models are full-featured models on their own and can be reused as instance models without the need to migrate them to a new metamodel or exporting them to a different environment. Simultaneously, this shows a fundamental limitation of our approach and the principle of not generating an explicit metamodel: the conformance definition of instance models is wired into the validation and modeling logic of our implementation and, in contrast to a metamodel, cannot be exported to other environments. Therefore, using existing instance models in other tools requires to drop the constraints imposed from the example models, resulting in instance models that only conform to a generic graph metamodel. Lastly, the front-end of our approach is implemented using the Modelverse wrapper API and therefore communicates with the Modelverse over network sockets. This allows for a distributed deployment of front- and back-end.

5.3 Comparison with Existing Solutions

In chapter 2, we analyzed existing solutions with respect to the following four areas:

1. **Unconstrained Input**: How are example models created and how much is the user constrained while expressing the requirements? What input methods are available? What visual variables can be used during this step?
2. **Metamodeling Support:** How are language constraints imposed and how much user intervention is required?
3. **Co-Evolution:** How is the co-evolution problem solved? Does a systematic classification exist?
4. **Tool Support:** Does an implementation exist? Was it evaluated? Is it integrated in a metamodeling tool or a standalone application?

In the following paragraphs, we elaborate on how our approach supports these areas and point out differences and similarities to the other solutions. A particular focus lies on comparing our approach, where all language constraints are computed directly from the example models, to solutions such as *metaBup*, where an explicit metamodel is generated.

5.3.1 Unconstrained Input

Our implementation provides a user interface that allows to create visual example models by drawing them on a canvas. The available drawing primitives are basic, but sufficient to capture a wide range of possible symbols. To map the underlying ATG data structure as closely as possible, we constrained our approach to connection-based languages. Therefore, it is possible to sketch example models with the visual expressiveness of figure 2.2, since entities and their relationships can be expressed by nodes and edges. Furthermore, our approach supports attributing these entities. However, spatial relationships between drawn symbols as implemented in *metaBup* are not considered. Also, our implementation does not feature any kind of sketch recognition, which is present in tools such as *Scribbler* and *FlexiSketch* and could speed up the sketching process.

Compared to other solutions, we highlight the flexibility of our approach: it is possible to design a different input method for example models by exchanging the front-end. Support for richer visual variables such as color or line width can be added by modifying the concrete syntax metamodel accordingly. Even purely textual example models can be supported as long as they conform to the ATG metamodel. In fact, adding textual example models can be done by uploading them as a text file to the Modelverse, using an already existing command line prompt. Therefore, we see our implementation close to the *Model Workbench*, which supports similar input methods, with the only difference that we do not support processing spatial relationships. This is reflected in table 5.1, which aligns the input capabilities of our approach with the other investigated tools.

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)
Moodling	Textual and visual	Freehand	Yes	Yes	No

Table 5.1: Tool support regarding example model input capabilities with our approach for comparison

Tool	Language Constraints	Advanced Meta-Constructs	Automation	MDE activities
Scribbler	Manual metamodel definition	None	Manual	Supported by EMF
MLCBD	Implicit metamodel generation	None	Semi	None
FlexiSketch	Implicit metamodel generation	None	Full	None
metaBup	Explicit metamodel generation	Inheritance, abstract classes, compositions	Full	Supported by EMF
Model Workbench	Explicit metamodel generation	Inheritance, abstract classes	Semi	Not assessable
Moodling	On the fly construction	None	Full	Supported by Modelverse

Table 5.2: Overview of metamodeling capabilities with our approach for comparison

5.3.2 Metamodeling Support

Table 5.2 classifies the metamodeling capabilities of our approach according to the criteria already used in 2.2.2. Similar to *MLCBD* or *FlexiSketch*, our approach does not generate an explicit metamodel. All language constraints are inferred from the example models on the fly and in a fully automated manner. This increases the flexibility, reduces the cognitive load and simplifies co-evolution, but decreases the performance and scalability, since for every modeling operation, the constraints have to be computed again. Our implementation supports an theoretically unlimited amount of example models and is implemented in a metamodeling environment, thus all models can directly be used for MDE activities such as transformations, analysis and code generation. In particular, this includes example models as well. As remarked in 3.2.1, model transformations that rely on graph pattern matching and consist of a LHS to match, a RHS to replace and a NAC as an apply condition require to construct a RAMified metamodel for the patterns themselves. Although our approach does not construct an explicit metamodel, a language for the patterns can be obtained by RAMifying the ATG metamodel, to which both instance- and example models conform to. The drawback here is the generality of the ATG metamodel, which results in a generic pattern language. As a result, it is possible to construct patterns with elements that do not appear in any model. For instance, nodes in a pattern can have types that do not appear in any example model. This is a contradiction to the principles of MDE, where languages should be as constrained as possible to prevent mistakes during modeling.

Advanced constructs such as inheritance, abstract classes and compositions, which are commonly found in metamodels that are based on class diagrams are not supported, as they require the user to give additional hints about such notions and therefore require metamodeling expertise. Implementing such constructs in our approach is possible by adapting the conformance relationship accordingly and providing the possibility to give the required hints via e.g. annotating nodes in the user interface, similar to *metaBup*. Furthermore, the issue of computing multiplicities only from examples remains unsolved: While it is possible to determine exact multiplicities by setting the bounds to the minimum and maximum cardinality of all example models for every element, the user's intent might have been a different one. This becomes especially apparent for high multiplicities that only can be implicitly expressed by adding the corresponding amount of elements to an example model. Therefore, this problem can only be solved on meta-level. For our approach, we imagine a functionality similar to *FlexiSketch*, where the user has the possibility to edit the multiplicities by hand.

Further limitations or our approach lie in the possibility to model additional constraints for the DSML. For instance, UML defines the *Object Constraint Language* (OCL) to apply rules to models in a declarative way [91]. Similar features can be found in the Modelverse, where the action language can be used to impose additional constraints upon instances. As a result, it is possible to further reduce the set of valid models of the DSML. A typical example is the number of tokens in a Petri net place, where only positive numbers and zero should be allowed. Such a constraint is difficult to model without using some sort of declarative language that restricts the token attribute of a place. Currently, our approach does not exhibit any functionalities to model such constraints as one principle is to hide as much metamodeling activities from the user as possible. While it is feasible to allow the definition of constraints on the level of example models, various changes are required: first, a type system for attributes needs to be introduced, alongside with redefining an attribute as a tuple of a key and a (typed) value. Furthermore, an input method for the declarative constraint language needs to be implemented in the front-end. With these changes, imposing further constraints on node attributes would be possible. Global constraints, such as refining the number of allowed nodes in an instance model suffers from the fundamental issue that example models are instance models of a metamodel themselves and therefore are affected by the constraints they would define. This could be solved by verifying the constraints only for models that are not example models.

5.3.3 Co-Evolution

In our approach, the example models are elevated as the primary descriptive element of the DSML. As a direct consequence, only forward evolution is supported. Again, this reduces the cognitive load and is more intuitive for users without metamodeling experience, since it does not require them to work on the level of metamodels. All possible changes are systematically classified in regard to their effects on the conformance relationship between example- and in-

stance models. Furthermore, all breaking changes are automatically resolvable by applying equivalent operations on the broken instance models. Thus, it is possible for the user to change the language at any point in the process and let the instance models co-evolve without manual intervention. We provide the possibility to execute structural changes locally on one example model or globally on all example models simultaneously. The reason for this is to avoid manually editing all example models and applying the same change all over again. This mitigates an inherent disadvantage of a purely example-model-based approach, where the meta-level is not exposed to the user: by introducing global operations, it is possible to perform changes as if they would be performed on the metamodel of a language. In particular, this is effective for language refactorings, such as renaming or deleting elements. In that sense, global operations mimic their corresponding operations on meta-level.

In 2.2.3, we used a set of test cases to assess the co-evolution capabilities of the investigated tools. In detail, a resolvable and an unresolvable scenario was constructed for both backward and forward evolution. Our approach only supports forward evolution, but is capable of handing both the resolvable and unresolvable scenarios fully automated, as we will demonstrate now. Figure 5.1 shows an instance model that conforms to the two example models shown in 5.2. When performing a global rename of the node "Router" to "DSL-Router", the rename is executed on all example models and successively on all instance models that have a node that is typed by "Router". The updated instance model can be seen in figure 5.3. Alongside the node in the model, also the list of available types was updated accordingly to show the retyped element.

In a similar fashion, the evolution scenario depicted in 2.7, which is referred to as unresolvable in the scope of metamodel-centric approaches, can be handled by our approach. Based on the same instance and example models of 5.1 and 5.2, the following operations on the example models result in an automated evolution of the instance model:

1. Global *add node* operation with type "Switch". This makes the type mandatory and automatically adds it to the instance model.
2. Global *delete edge* of edge between "Router" and "PC" in example model 1 and between "Router" and "AP" in example model 2. This results in the edges being deleted in the instance model as well.
3. Global *add edge* between "Router" and "Switch". This adds the same edge to the instance model.
4. Global *add edge* between "Switch" and "PC" as well as between "Switch" and "AP". This adds the same edges to the instance models, since the edges became mandatory.

The resulting instance model with the added "Switch" element is shown in figure 5.4.

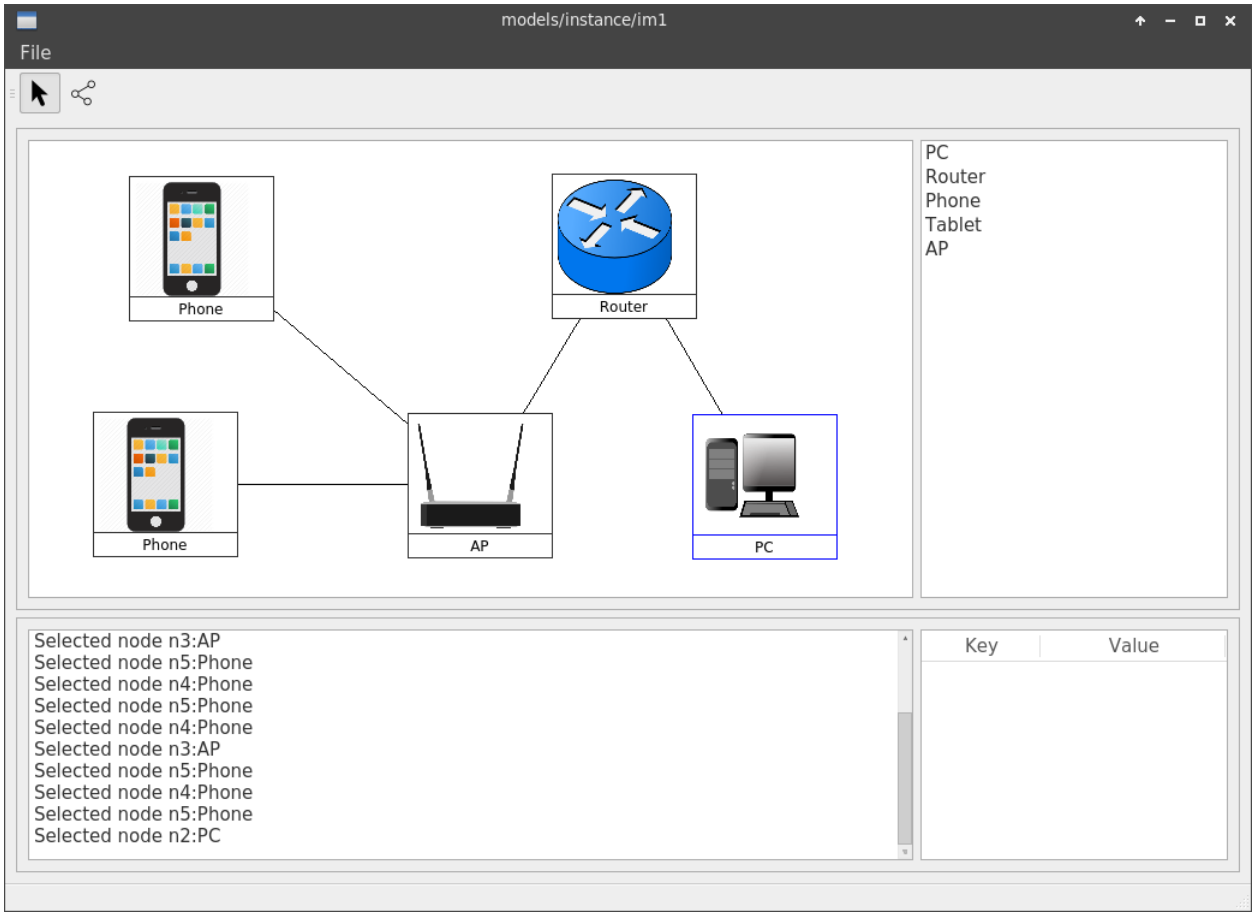
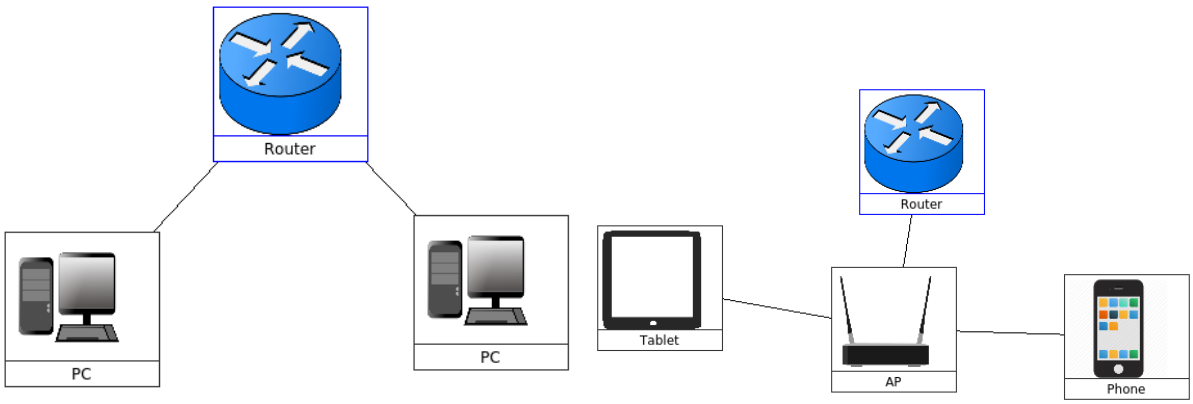


Figure 5.1: The instance model prior to retyping a node



(a) Example model 1 (b) Example model 2

Figure 5.2: Example models prior to retyping a node

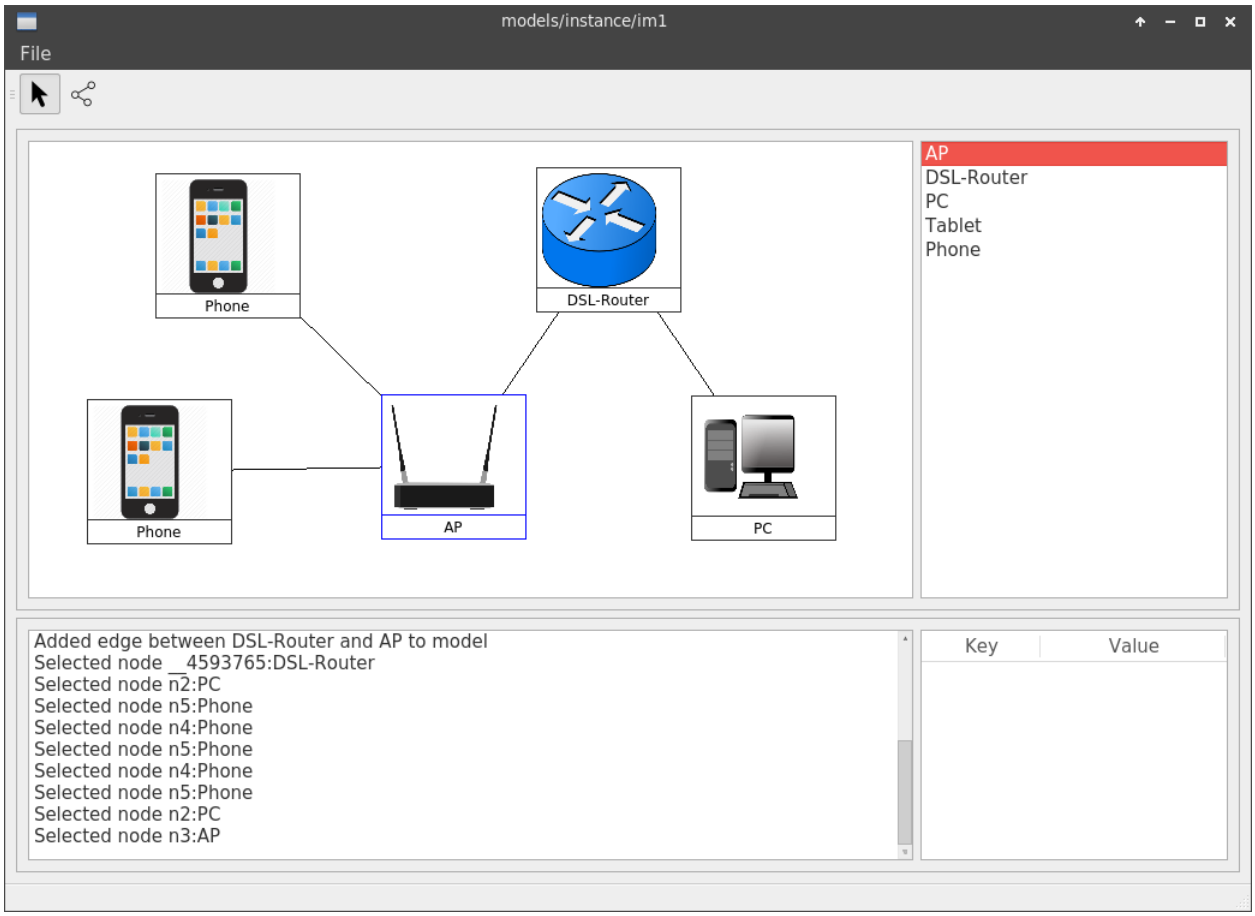


Figure 5.3: The instance model after retyping the “Router” element in an example model

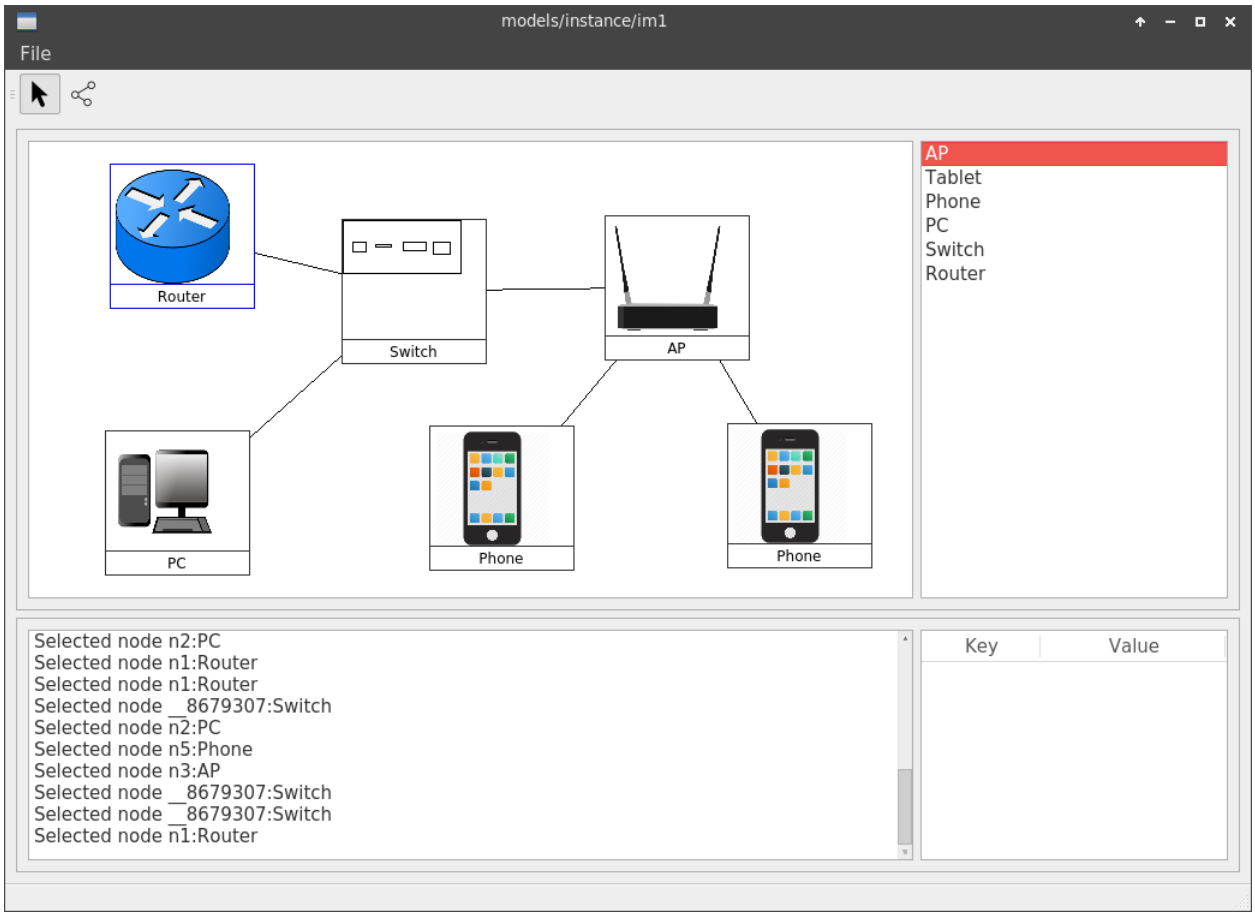


Figure 5.4: The instance model after adding a mandatory “Switch” node between the “Router” and all its connecting nodes

Furthermore, also the concrete syntax can evolve. This is possible since the concrete syntax representation of each type is stored as a model itself, alongside with the ATG models for the example- and instance models. It is thus possible to replace inadequate concrete syntax symbols by changing the corresponding concrete syntax model. This can be done by either sketching a new set of primitives and typing the group by the existing type that should be replaced or by uploading an actual image file as a concrete syntax model. For the latter, we have implemented a command-line script that reads an image file and performs the necessary Modelverse API calls to upload a concrete syntax model with the image file as data. After overwriting an existing concrete syntax model, the new representation for a type can be and used immediately in both the example- and instance modeling phase. To our knowledge, the only other tool that supports similar features is *FlexiSketch*, where multiple representations for the same type can be sketched and overwritten.

As a summary, Table 5.3 compares the co-evolution features of our approach with existing tools. Although backward evolution is not possible by design, we provide global operations to make structural changes to the language less cumbersome, especially with a large amount of example models. All evolution scenarios are classified according to their effects on the conformance relationship and can be resolved fully automated.

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
Moodling	Yes	No	Effects on conformance

Table 5.3: Summary of co-evolution features with our approach

5.3.4 Tool Support

Table 5.4 summarizes the tool support of our approach based on the evaluation in section 2.2.4. We implemented our approach as a front-end for the Modelverse, a multi-paradigm metamodeling tool. Therefore, all models are contained in a full-featured metamodeling environment

and the need to switch between different tools such as in *metaBup* is eliminated. However, the usability of our tool was not evaluated in a user study. Furthermore, due to the on-the-fly construction of the language constraints, the performance is expected to decrease exponentially when the size of the example model set increases. Additionally, no collaboration support was implemented in the tool. However, the Modelverse allows for multiple parallel users and implements a permission system with a notion of owners and groups. Therefore, it is possible to extend our approach by features similar to *FlexiSketch*, where models can be shared between different users and multiple users can sketch example models together. This includes the possibility to lock the example modeling phase, which can be done easily in our approach since example modeling and instance modeling are clearly separated. Lastly, we want to mention that, although the process is designed to switch between the unconstrained example modeling and constrained instance modeling phases, it is possible to never advance to the instance modeling phase and construct all models during example modeling exclusively. In contrast to other solutions where example models exist in a different environment, our example models are full-featured models on their own and can be used in further MDE activities just like instance models.

5.3.5 Result

In section 2.2.5, we presented the result of the initial evaluation of existing tools as a radar chart with the four different key areas *Unconstrained input*, *Tool support*, *Metamodel generation* and *Co-evolution* as axis. Figure 5.5 shows the same chart with the result of the evaluation of our approach as a dashed line.

The weakest area of our approach is the tool support. Albeit implemented, the tool is in a prototypical state and requires usability and performance improvements in order to offer a satisfactory user experience. This includes responsiveness and interactive feedback, since operations such as the automated co-evolution of instance models are currently executed silently in the background. Additionally, no user study was conducted. Such a study could further investigate the required experience, cognitive load and user friendliness of our approach, especially compared to solutions that infer the language constraints by explicitly generating a metamodel.

Unconstrained input is related to the tool support: we have implemented a simple UI for example model sketching, where arbitrary shapes can be constructed with geometric primitives such as lines or rectangles. However, no support for additional visual features such as color or line width is present, thereby limiting the visual expressiveness of the concrete syntax. However, we support connection-based visual languages that with graph-like structures and allow for annotations that function as attributes. Finally, no sketch recognition was implemented, as new types are registered immediately and can be instantiated thereafter by double-clicking on the corresponding item in the list of types.

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

Table 5.4: Summary of tool support evaluation

The metamodeling support of our approach is on the same level as *metaBup*, since we implemented our tool with the Modelverse as back-end. Therefore, all MDE activities that are supported by the Modelverse can be directly applied to instance- and example models. Furthermore, example models can be reused as instance models, since they both conform to the same ATG metamodel. One major difference to all other tools is how the language constraints are computed: instead of generating a metamodel to define the structure of the language, the constraints are inferred on the fly during instance modeling from the example models. Although this requires a re-definition of the conformance relationship between instance models and the abstract syntax given by the example models, it removes the metamodel generation as an intermediate step in the process and increases the agility by encouraging short development iterations and rapid feedback.

Lastly, our approach has throughout support for model co-evolution. By removing the intermediate metamodel generation step, the language can only be evolved by changing the example models. This reduces the cognitive load as it hides meta-level concepts from the user. All possible example model changes are classified with regard to their potential effect on the conformance relationship to the instance models. Furthermore, all changes can be resolved automatically by applying the same model transformation that was used to change the example model(s) to the instance models. Therefore, all instance models evolve automatically as the language changes. Compared to other solutions, the example models serve as the primary description of the abstract syntax throughout the whole process, even when the language development has stabilized and instance models have been created. This is in contrast to tools such as *metaBup* that explicitly generate a metamodel and where the metamodel eventually replaces the example models. As a result, forward evolution carries the risk to break instance models, since the example and instance models exist detached from each other.

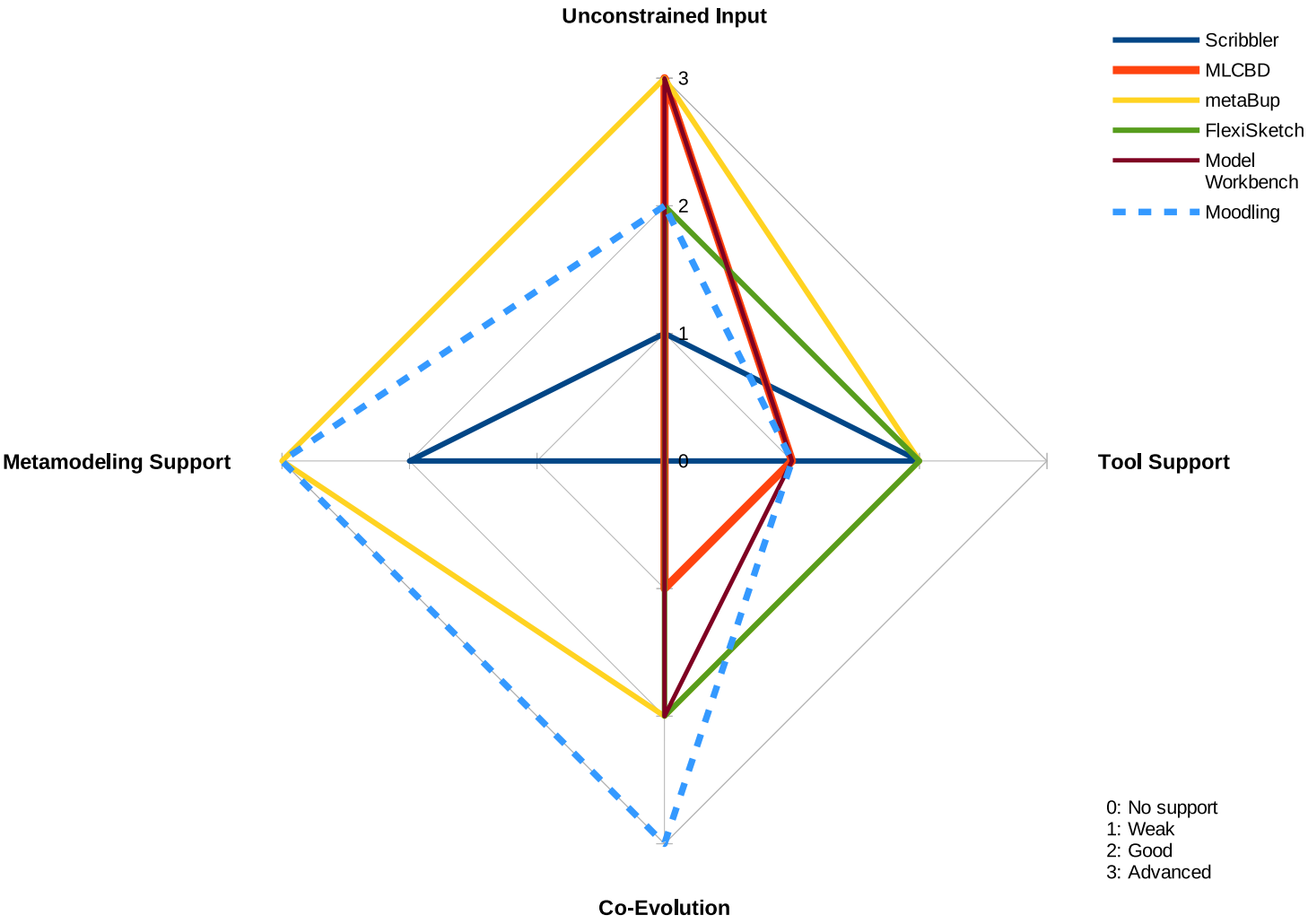


Figure 5.5: Summary of strengths and weaknesses of solutions, including our approach

Chapter 6

Conclusion

This thesis presented an approach for domain-specific language design by example. In chapter 1, an introduction to model-driven engineering in general and example-driven language design in particular was given. Chapter 2 elaborated on related work by systematically analyzing and comparing existing solutions for the problem. The theoretical concepts of our approach were discussed in chapter 3. In detail, it was shown how example models can serve as prescribing elements for a domain-specific modeling language and how they can be used for an agile and integrated design process with focus on short iterations and incremental language design. Chapter 4 presented a prototypical implementation of our approach with a self-modeled multi-paradigm metamodeling environment as basis. This implementation served as a proof for the feasibility of our approach. It was evaluated and compared to existing solutions in chapter 5.

6.1 Summary and Contributions

In conclusion, an agile and integrated DSML design process was developed. It has been demonstrated that it is possible to use example models as the primary description of a language. This eliminates the need to design a DSML on meta-level and makes the process accessible for users without in-depth language engineering experience. All language constraints are inferred directly from the example models. Furthermore, we showed how the language can be incrementally developed by adapting the example models to reflect new or changed requirements, while letting the instance models co-evolve in a fully automated manner. Additionally, we explicitly modeled the concrete syntax in our approach. This makes it possible to evolve the representation of a model alongside with the abstract syntax, which is of particular importance since initially, example models and the concrete syntaxes of language concepts are sketched by hand and might require iterative refinement during the process.

All aspects of our approach were implemented in the Modelverse, a multi-paradigm metamod-

eling tool. We provide a user interface, which allows for model creation and manipulation. This user interface can operate in either an unconstrained example modeling mode, where requirements for the DSML can be expressed by sketching example models, and in a constrained instance modeling mode, which uses the example models to constrain the user in the available modeling activities. Since all our models reside in the Modelverse, they can be reused for further MDE activities without the need to migrate them to a different environment. This also includes example models, since, in our approach, they do not differ from instance models on a conceptual level. Therefore, our implementation allows to use example models as instance models and vice versa.

6.2 Limitations and Future Work

Future work can be conduced in a plethora of directions:

- **Expressible Languages** During the design of our approach, we limited ourselves to connection-based languages which can be expressed by a graph data structure. In particular, it is currently not possible to model edges without using additional nodes, since edges are always untyped and undirected. This limits the user's freedom, since edges with a notion of type and direction are common elements in modeling languages. We believe that our approach is flexible enough to accommodate for such extensions. The ATG metamodel, to which example- and instance models conform to, inherits the flexibility of the class diagram formalism of the Modelverse and therefore, the edge association can be modeled as class with a type and direction. The concrete syntax metamodel is expressive enough to store different edge representations, since it is capable of describing all geometric primitives by lines. However, the user interface needs to be extended to render different kinds of edges, according to the concrete syntax metamodel. Additionally, it must provide a method to sketch the different kind of edges and give the user the possibility to annotate them, according to the type and direction properties.
  Additional visual variables present in other approaches, such as the spatial relationship of symbols or color were not investigated in this thesis and can be subject of future research as well. This is especially interesting with regard to object-oriented concepts such as inheritance, which could be expressed by a visual containment relationship.

- **Phase Separation** We strictly divide the process of our approach into an example- and instance modeling phase and therefore make a clear distinction between an open and a constrained environment for creating models. This means that whenever an operation is invalid during instance modeling but should be allowed, the corresponding adaptions have to be made by going back to the example modeling phase. We anticipate a frequent occurrence of such a scenario, since, especially at the beginning of the process, the language undergoes many revisions and requires steady changes to appropriately express the

aspects of the system to model. Therefore, an alternative to switching back and forth between the language design and usage phases could be to automatically introduce the required changes in the example models upon user request without leaving the instance modeling environment. Essentially, this would blur the currently existing strict separation between the two phases, since changes to the language could be performed during instance modeling as well. However, we see the danger of providing an unconstrained modeling environment during instance modeling, which gives the user too much freedom and increase the possibility to make mistakes. This issue could be addressed by using a lock mechanism for the language development: when the lock is active, all changes that alter the language are disallowed and therefore, the language can not evolve anymore. Only when this lock is explicitly removed, changes to the language become possible again.

- **Requirements Engineering** In our approach, the language requirements are directly expressed by sketching example models. Therefore, no formal specification of the requirements exists. As a result, both language testing and collaboration among multiple users is difficult, since they might have differing ideas on how the language must look like. Without a preceding requirements engineering phase, reaching consent based solely on a set of example models is a possible source of conflicts and increases the risk of miscommunication between the users. Since we aim for an agile approach for example-driven DSML design, it could be investigated how methods from agile software development processes can be used in this context.

- **Performance** The performance of our implementation is currently rather low, mostly because of the Modelverse. During instance modeling, delays are particularly noticeable since every operation has to be validated prior to its execution. During example modeling, delays are introduced due to the language evolution handlers, which are called after every change and keep the instance models in conformance. However, our approach is independent from the underlying metamodeling environment and therefore can be implemented with more efficient tools. Furthermore, we believe that the performance can benefit from implementing key parts of our approach such as the instance modeling operator validation and the conformance relationship directly in the Modelverse instead of in the front-end.

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] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *International Conference on Graph Transformation*, pages 402–429. Springer, 2002.

[6] Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, et al. State Chart XML (SCXML): State machine notation for control abstraction. *W3C working draft*, 2007.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] Colin De La Higuera, Jean-Christophe Janodet, Émilie Samuel, Guillaume Damiand, and Christine Solnon. Polynomial algorithms for open plane graph and subgraph isomorphisms. *Theoretical Computer Science*, 498:76–99, 2013.

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

[28] 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.

[29] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *International conference on graph transformation*, pages 161–177. Springer, 2004.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. Using graph transformation for practical model-driven software engineering. In *Model-driven Software Development*, pages 91–117. Springer, 2005.

[40] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution*, pages 3–1. IEEE, 2007.

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

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

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

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

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

[46] 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.

[47] 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.

[48] Lucas Heer. Sketch-based metamodel construction: A literature review. 2018.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Andreas Kästner. On the Transformation from Incomplete Object Diagrams to Incomplete Class Diagrams. Master's thesis, University of Bremen, Bremen, Germany, 2017.

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

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

[57] Alexander Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice Satellite Workshop of MODELS*, page 166, 2005.

[58] Thomas Kühne. What is a model? In *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*, 2004.

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

[60] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In *International Conference on Model Driven Engineering Languages and Systems*, pages 240–255. Springer, 2009.

[61] 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.

[62] Jill H Larkin and Herbert A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987.

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

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

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

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Bart Meyers, and Hans Vangheluwe. The formalism transformation graph as a guide to model driven engineering. *School of Computer Science, McGill University, Tech. Rep. SOCS-TR2012*, 1, 2012.

[72] Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. FTG+PM: an integrated framework for investigating model transformation chains. In *International SDL Forum*, pages 182–202. Springer, 2013.

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

[74] 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.

[75] 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.

[76] 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.

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

[78] 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.

[79] Tom Mens. On the use of graph transformations for model refactoring. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 219–257. Springer, 2005.

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

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

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

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

[84] 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.

[85] Daniel Moody. What makes a good diagram? improving the cognitive effectiveness of diagrams in is development. In *Advances in information systems development*, pages 481–492. Springer, 2007.

[86] 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.

[87] Daniel Moody and Jos van Hillegersberg. Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. In *International Conference on Software Language Engineering*, pages 16–34. Springer, 2008.

[88] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004.

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

[90] Sadaf Mustafiz, Joachim Denil, Levi Lúcio, and Hans Vangheluwe. The FTG+PM framework for multi-paradigm modelling: An automotive case study. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pages 13–18. ACM, 2012.

[91] Object Management Group (OMG). Object Constraint Language (OCL) Specification, Version 2.4. OMG Document Number formal/2014-02-03 (`http://www.omg.org/spec/OCL/2.4`), 2014.

[92] Object Management Group (OMG). Meta-Object Facility (MOF) Specification, Version 2.5.1. OMG Document Number formal/2016-11-01 (`http://www.omg.org/spec/MOF/2.5.1`), 2016.

[93] 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.

[94] 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.

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

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

[97] Rodrigo C. O. Rocha. Typed graph theory: Extending graphs with type systems. Preprint available at `http://rcor.me/papers/typed-graph-theory.pdf`.

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

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

[100] 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.

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

[102] 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.

[103] 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.

[104] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[105] 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.

[106] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

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

[108] Eugene Syriani and Hans Vangheluwe. A modular timed graph transformation language for simulation-based design. *Software & Systems Modeling*, 12(2):387–414, 2013.

[109] 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.

[110] Simon Van Mierlo. *A Multi-Paradigm Modelling Approach for Engineering Model Debugging Environments*. PhD thesis, University of Antwerp, 2018.

[111] Simon Van Mierlo, Bruno Barroca, Hans Vangheluwe, Eugene Syriani, and Thomas Kühne. Multi-level modelling in the modelverse. In *Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014): September 28, 2014, Valencia, Spain/Atkinson, Colin [edit.]; et al.*, pages 83–92, 2014.

[112] Yentl Van Tendeloo and Hans Vangheluwe. Explicitly modelling the type/instance relation. In *Proceedings of MODELS 2017 Satellite Event*, pages 393 – 398. Ceur-WS, September 2017.

[113] Yentl Van Tendeloo and Hans Vangheluwe. Linguistic conformance check (as implemented in the modelverse). MDE lecture notes, `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/conformance_algorithm.pdf`, 2017.

[114] Yentl Van Tendeloo and Hans Vangheluwe. The Modelverse: a tool for multi-paradigm modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, pages 944 – 955. IEEE, December 2017.

[115] Hans Vangheluwe, Juan De Lara, and Pieter J. Mosterman. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the AIS'2002 conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal*, pages 9–20, 2002.

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

[117] 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.

[118] 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.

[119] William Winn. An account of how readers search for information in diagrams. *Contemporary Educational Psychology*, 18(2):162–185, 1993.

[120] 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.

[121] 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.

[122] 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.

[123] 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.

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

[125] Stéphane Zampelli, Yves Deville, Christine Solnon, Sébastien Sorlin, and Pierre Dupont. Filtering for subgraph isomorphism. In *International Conference on Principles and Practice of Constraint Programming*, pages 728–742. Springer, 2007.

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

Appendices

Appendix A

Source Code

The source code repository for the implementation can be found at:
`https://msdl.uantwerpen.be/git/lucas/modelverse`

Appendix B

Sample Process

Listing B.1: Implementation of the Moodling process in the Modelverse Process Model formalism

```
Start start {}
Finish finish {}

Exec new_exm {
    name = "process/new_exm"
}

Exec edit_exm {
    name = "process/edit_exm"
}

Exec query_another_exm {
    name = "process/query_another_exm"
}

Exec new_im {
    name = "process/new_im"
}

Exec edit_im {
    name = "process/edit_im"
}

Exec query_revise_lang {
    name = "process/query_revise"
}

Data exm {
```

```
    name = "models/example/ex1"
    type = "formalisms/graphMM"
}

Data im {
    name = "models/instance/im1"
    type = "formalisms/graphMM"
}

Data consyn {
    name = "models/consyn/cs1"
    type = "formalisms/consynMM"
}

Decision another_exm {}
Decision revise_lang {}

Next(start, new_exm) {}
Next(edit_exm, query_another_exm) {}
Next(query_another_exm, another_exm) {}
Then(another_exm, new_exm) {}
Next(new_exm, edit_exm) {}
Else(another_exm, new_im) {}
Next(new_im, edit_im) {}
Next(edit_im, query_revise_lang) {}
Next(query_revise_lang, revise_lang) {}
Then(revise_lang, edit_exm) {}
Else(revise_lang, finish) {}

Produces(new_exm, exm) {
    name = "example_model"
}

Consumes(edit_exm, exm) {
    name = "example_model"
}

Produces(edit_exm, exm) {
    name = "example_model"
}

Produces(edit_exm, consyn) {
    name = "concrete_syntax"
}

Produces(new_im, im) {
    name = "instance_model"
}

```

```
78 Consumes(edit_im, im) {
79     name = "instance_model"
80 }
81
82 Consumes(edit_im, consyn) {
83     name = "concrete_syntax"
84 }
85
86 Produces(edit_im, im) {
87     name = "instance_model"
88 }
```