Frustration by construction? - Exploring relaxed conformance in a modular graphical model editing environment

Master Thesis Proposal

1 Introduction and Motivation

Models, and with them their graphical representations, are an essential part of modern software development. They serve as an alternative to textual program code for viewing and creating software systems, both as a pure documentation tool (to provide an abstraction of a complex system for a certain end) and, in paradigms such as Model Driven Engineering (MDE) and Business Process Management (BPM), as a self-sufficient way and a primary artifact to create entire systems. In particular, so-called Domain Specific Languages (DSLs), a term used to describe modeling languages specifically tailored to a certain field of application, have been described as a way to move thinking during the system design phase from the *solution space*, i.e. the technologies in which a system will be realized, to the *problem space*, putting emphasis on the needs to be fulfilled by the system in the future [1], and making the system verifiable against these needs [2]. DSLs also enable domain experts to be a more integral part of the system development process, as they put systems in terms understandable for them, especially when using intuitive graphical notations [3]. As a result of applying MDE in practice, it has been shown that there can be tremendous increases in productivity, result quality and learnability, for example in [4].

Already when modeling purely for documentation purposes but especially when applying MDE, graphical modeling tools become an essential part of system engineers' workflows, akin to text editors and text-based IDEs when developing with a traditional code-only approach. The intuitive, creative and dynamic nature of graphical notations is leveraged best if the modeling software itself is pleasant and efficient to use, supporting the diverse ways and mindsets in which a developer may approach a problem. Furthermore, as both the general technical environment and the graphical notations themselves are often dynamic and in continuous development, it is important to have an easy way of extending, modifying and adapting tools both to varying conditions and to changing requirements.

One possible, rarely investigated issue with existing tools is the strict and implicit enforcement of the syntactical structure defined by the modeling language, making users unable to create incorrect models even as a work in progress and forcing them to structure their workflow around the tool and the notation used, not vice versa. Traditional "top-down" approaches have been centered around the metamodel, that means the construct which defines what elements can be contained in the model and how they may be related. Modeling processes were then working on the premise that the way to achieve a correct model is enforcing adherence to this metamodel at all times [5]. Therefore, in the space of graphical notations, model syntax is often hard-coded as an integral part of the purpose-built editing frontend component

[6], also creating a problem whenever a change needs to be made to the tool's capabilities. Need for such changes can come from the environment on the one side, such as needing editing capabilities on new platforms or the requirement for seamless collaboration on a single model [7]. Also, the languages themselves can demand flexibility, like varying notations that need to be interpreted with a common abstract syntax in mind [6] or changing metamodel structures while the modeling process is still in its earlier phases [5], [8].

2 Problem Statement

As tools should be designed to support workflows and not vice versa, graphical modeling environments should accommodate the typical process of model development, which goes through various degrees of formality as shown in [5]. As an example, consider a user designing a model in an already well-defined language like the UML class diagram notation. One may start out with an informal and incomplete model, omitting attributes like edge end multiplicities and leaving dangling edges where the element that will be target of the edge is not yet clear. These draft models can then serve as a starting point for ideation, communication and discussion, and can be refined to become more and more adhering to formal requirements as the system design process goes on. In the end, the result would be a well-defined model that can be used for code generation, verification and other MDE activities that need rigidly defined models.

Modeling language definitions generally consist of two different areas, which are equally important for their practical usage: The *abstract syntax* and the *concrete syntax* [9], [10]. The former is the domain of the *conceptual elements* a modeling language includes and how they can have relations to each other in a valid instance of the modeling language. Which semantic elements exist for use in modeling and how they may be related is defined by an *abstract syntax metamodel*, again described in terms of a meta-metamodel, usually consisting of generally understood terms like object-oriented programming classes and thus describable with generic representation mechanisms [9]. Concrete syntax, in comparison, defines how the concepts of the abstract syntax are presented to the user. The metamodel of the concrete syntax is a representation language, i.e. a visual language consisting of the elements available for model construction as well as the rules on how those elements may be used to construct a valid instance (called *sentence*) of this visual language [11]. In visual language research, these concrete syntax metamodels are sometimes again referred to as the "abstract syntax" [12] of the visual language, which is misleading in this context, as the visual language itself is strictly a concrete syntax construct and multiple independent concrete syntaxes can exist for one single abstract syntax [6].

Flexible approaches, for example proposed in [5], [8], [13], solve the aforementioned challenge of tightening conformance requirements over the course of a process from the *abstract syntax* point of view. That means, they support the checking of a given model in various degrees of strictness, either considering the modeling process explicitly [5], or implicitly via disabling conformance checking for certain metamodel aspects [13].

However, these approaches do not yet state how this relaxed conformance is to be represented and implemented in the concrete syntax editing environment, or only consider text-based notations [5]. Furthermore, the relaxed conformance at the abstract syntax level is in a way independent from the concrete syntax level – even if the conformance is strictly enforced at the abstract syntax level, users may desire not to enforce the strictness in the *transition* from concrete to abstract syntax until they have actually finished their current editing task and want to proceed using the model for operations that require complete and correct abstract syntax.

As an analogy, rigid conformance graphical editors can be likened to an IDE for a traditional text-based programming language that only allows the user to insert well formed, complete blocks of code and edit them only in ways that are guaranteed not to produce a syntax error. Instead, in text-based software

development, developers expect being free to enter any text coming to their mind, with modern tools trying their best to provide helpful suggestions for fixing problems or for executing actions likely to be useful for the task at hand.

3 Purpose of the study

In this thesis, a concept and an accompanying demonstrating prototype is to be be built and evaluated that strives to explore an approach similar to the one used by aforementioned text editors, dubbed "relaxed conformance visual editing". Building on the architecture concept of Van Tendeloo et al. [6] based upon clear separation of concrete and abstract syntax, a modular, decoupled framework is designed to make it possible to create highly dynamic and flexible editing environments for graphial modeling notations of all kinds, while also retaining the capability of the tool to "understand" the syntax and meaning behind the model, and providing the user with helpful assistance in efficiently creating correct models.

This development should result in an executable prototype of an editor in which models of a simple notation can be created. The editor should allow freely creating diagrams without limiting user interaction because of the boundaries set by the abstract syntax, while still offering helpful, language specific assistance. In essence, it should be a blend of an editor that can be used for Model Driven Engineering with documentation-focused, free approaches found in business diagramming applications like Microsoft Visio. More precise information about the editor's features can be found in section 6.2.

The focus lies in the conceptual structure of the system and its applicability for a wide range of graphical modeling languages (as opposed to high end-user usability for a specific modeling language). The architecture should allow the reuse of its components and structure, for example to develop relaxed conformance editing components for other notations and languages, automatic editor generation features or use the application for other research areas such as model sharing.

As a summary, the goal of this thesis is to explore a possibility to implement non-strict conformance in a graphical model editor while on the other hand providing users with enough help to be able to create conforming and machine-processable models. Consequently, the resulting artifact is to be used to evaluate the usefulness of the approach in respect to the explained goals.

4 Review of the literature

This thesis is related to and based on other, currently ongoing research in multiple fields. This section provides an overview of literature serving both as the basis of the developed approach and an important input as to why the proposed concept may be worth pursuing. For the sake of brevity, an excerpt of the most relevant literature will be shown; some others were cited in the other sections, especially 1 and 2, and will be elaborated further in the thesis itself.

4.1 Application data and architecture structure foundation

The reasoning behind the structures outlined in section 7 is drawn from different sources in the fields of Domain Specific Languages and Visual Languages. The most important ones are:

4.1.1 Y. van Tendeloo, S. van Mierlo et al., Concrete syntax: a multi-paradigm modelling approach [6]

Summary The authors of this paper criticize the lack of flexibility in regard to concrete syntax in the currently available implementations of modeling tools. Specifically, they find that generally, the representations (concrete syntax) and the concepts behind them (abstract syntax, the definition of

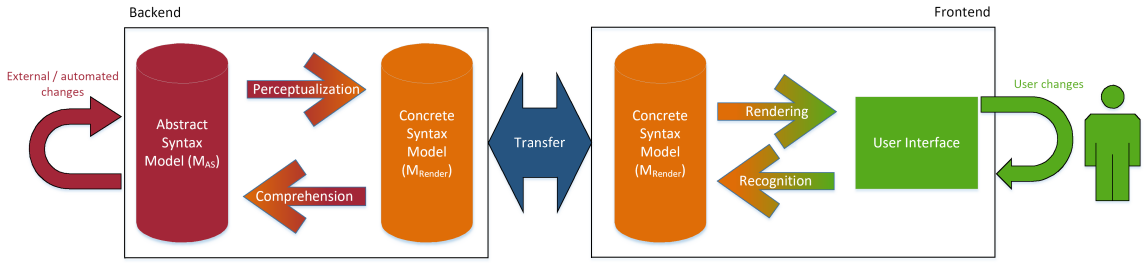

Figure 1: Decoupled editing process (own graphics based on [6])

which is commonly referred to as a *metamodel*) are too tightly coupled. In these classic approaches, the modeling frontend is one monolithic component where both the graphical display and the handling of abstract syntax – the mapping from a model instance to the metamodel backing it – is managed. It is claimed that this leads to high implementation effort for supporting new features in languages, often realized by custom plug-ins. Furthermore, it limits the availability of alternatives in the modeling application – often, even small changes to the representation that have no effect on the abstract syntax are difficult to implement because of the strong relation between concrete and abstract syntax parts. Apart from that, the support for defining specialized operations for domain-specific languages is not given.

As a solution, the study suggests an approach where concrete and abstract syntax are represented in metamodels which, in contrast to traditional approaches, are only loosely coupled by transformation process steps and primarily handled by separated components. The render metamodel contains the graphic elements of which a graphical representation (i.e. an instance of the concrete syntax) can be composed, such as graphic primitives (e.g. rectangles, ellipses). It is interpreted and instances of it manipulated by the frontend. On the other hand, there is the abstract syntax metamodel, which contains the semantic structures of a model, i.e. the concepts that are actually represented with the modeling language. This part of the model is managed by an independent backend. Transfer of model information from frontend to backend and vice versa has four different activities: The abstract syntax model (expressed in the concepts of the abstract syntax metamodel) is *perceptualized* by the backend, that means converted to the target render metamodel. The render model is then *transferred* to the frontend, which can now *render* the given model. When the user makes changes in the rendered model instance (if the frontend allows changes), the changes are *recognized*, that means the user's operations in the user interface are applied to the representation of the concrete syntax. The changes are then *transferred* to the back-end, where the *comprehension* activity maps the concrete syntax changes back to the abstract syntax. Figure 1 shows the schematics of the activities involved in the editing process.

Through this architecture, the rendering and user interaction is decoupled from the abstract syntax and its metamodel. Consequently, multiple different frontends can be implemented, possibly on different platforms, as long as they share the same metamodel for concrete syntax and the same interface to the backend, which does not need to be adapted for each editor. Additionally, they envision multiple completely independent rendering forms for the same abstract syntax, by defining independent perceptualization components.

Relevance The work of Van Tendeloo, Mierlo et al. is central to the development of the concept developed in this project. The five step process proposed is the starting point for the architecture developed (albeit here, the steps are split up further), and the clear distinction between the frontend concrete syntax editing component and the backend managing the abstract syntax is kept as a

central building block.

Delimitation While the described article presents the basic editing process, the paper does not get into detail about the implementation in a real graphical editor. Furthermore, a more specific architecture has not been proposed, and nothing is said about how non-conformance is to be handled in the system. This thesis will build on the aforementioned fundamental concepts to build a graphical editing prototype with non-enforced conformance checking.

4.1.2 P. Bottoni, A. Grau, A Suite of Metamodels as a Basis for a Classification of Visual Languages

Summary In this paper, the authors describe various metamodels for the representation of the graphic syntax of modeling languages. They argue that the semantic interpretation of graphical representations requires a metamodel for the graphic notation. One notable aspect of this paper is the usage of the term *abstract syntax*, which differs from its meaning in [6] and thus might be confusing – as Bottoni and Grau concern themselves with the visual part only, their "abstract syntax" is actually closer to the term "concrete syntax metamodel" or "render metamodel" in [6], however with more information about the actual behavior of the elements attached. All metamodels are presented in UML class diagram notation, which makes them applicable to implementation in an actual system.

First, a base metamodel is defined, which contains graphic primitives (*GraphicElement*), grouped to an aggregate called a *ComplexGraphicElement*. These complex elements are then linked to an *IdentifiableElement*, representing a semantic element in the modeling language. These semantic elements can then be in a spatial relation to each other. For these relations, they distinguish between *direct* and *emergent* relations, the latter being determined by patterns / rules or a group of direct relations. Furthermore, the concept of *AttachZones*, where two elements can be linked to each other is introduced already in this most basic model, and direct spatial relations between identifiable (semantic) elements are derived from attachment between two of these zones among their graphical representations.

Furthermore, the class of *connection-based* models is introduced, which defines a *connection* as a type of element that *touches* an arbitrary number of identifiable elements. Constraints are attached to the connection through a *manager* class, determining whether the connection is directed or not, and how the multiplicities (number of elements to which the connection may be attached) on its different sides are. The class of connection based models is further specialized into representations for *graphs*.

Another different approach to language syntax is the class of *containment-based* languages, where a special class of direct relation consists of one element containing other elements. The authors take the containment-based class as an example for a spatially defined language, but also state that there are more ways to express relations through spatial arrangement, which were further described for example in [14].

Relevance This article, along with others from the field of Visual Languages (like [14]), serve as an important input dor designing the prototype's data model. Furthermore, the processing capabilities required to support typical features of visual notations can be derived. However, the concepts presented will have to be adapted to and merged with the ideas from [6] for the final design.

Delimitation The Visual Language theory papers provide definitions of the elements of graphical representations, but they do not give implementations of these ideas in actual software.

4.2 Other approaches developed in the field of relaxed conformance modeling

As mentioned before in section 2, there is already ongoing research about relaxed conformance, albeit these studies look at the topic from a *metamodeling* respectively *abstract syntax* point of view, whereas the proposed project comes from the *concrete syntax* side. However, these studies provide valuable insights in the requirements to consider for relaxed conformance editing and some of the benefits that can be expected from it. Furthermore, they give a perspective on what other use cases could be implemented in future work with the prototype to be developed.

4.2.1 E. Guerra, J. de Lara, On the Quest for Flexible Modelling [5]

Summary In this paper, the authors describe their implementation of a metamodeling toolkit that allows for varying levels of conformance. They argue that in the early phases of the modeling process, less strict conformance is required to support the development and discussion of ideas. As the models mature and get more precise, users will want to turn on specific conformance requirements until they arrive at a strictly checked model that is formal enough for use in Model Driven Engineering operations such as code generation or model transformation.

One unusual aspect of their approach is that the modeling process by itself is explicitly considered in the design of the proposed framework. On various stages of the process, different validation criteria are turned on and checked. The tooling also provides so-called quick fixes for inconsistencies and problems found in the models that are tailored towards the process at hand and its *intent*: When the goal of the process is the more traditional creation of a model from an existing metamodel (*top-down* approach), the software would suggest changing an incorrect attribute type to the one specified in the metamodel. On the other hand, for another process supporting the creation or evolution of a metamodel according to the example given by the currently edited model instance, the fix suggested would be to instead change the metamodel to fit the example.

Another noticeable trait is the flexibility given to the relation from model to metamodel. Here the ontological type (which is the semantic type of a model element, i.e. the link of a model element to the metamodel) is independent of the linguistic type (the type that the representation of a model element is given in the implementation language). This enables support for much more flexible typing or even model elements that are not (yet) typed.

Relevance The modeling process examples in the paper give good arguments on why relaxed conformance is worth pursuing and, in combination with the requirements explained by the authors, can serve as a basis for selecting editor functionality to be implemented. Furthermore, the proposed architecture and data structures (e.g. the aforementioned separation of linguistic and ontological typing) are to be considered when designing the prototype data model. In addition to that, the paper proposes the use of intelligent and situation-dependent "quick fixes" for the correction of non-compliant models after they have been freely edited, albeit only for text-based languages.

Delimitation While the editing functionality for text-based languages has already been implemented in their *KITE* system, it does not yet support graphical notations.

4.2.2 R. Salay, M. Checnik: Supporting Agility in MDE Through Modeling Language Relaxation [15]

Summary In this paper, the basic idea is postulated that relaxation of conformance requirements is very helpful for human interactions to be executed on the model, as opposed to automatic model operations such as transformation, which require strict conformance.

In contrast to [5], this article directly deals with situations that may occur in graphical concrete syntax, instead of concentrating on the abstract syntax and the conformance of textual concrete

syntax. The authors identify two distinct types of *agility*, i.e. ways of editing model elements that are not conform with the metamodel of the concrete syntax: *Omission agility* is the typical case of non-conformance – leaving required elements out in order to deal with or represent uncertainty, such as leaving an edge in a connection-based language unconnected to symbolize that it is not yet clear where this element belongs. *Clarity agility* is the introduction of additional information into the model which is out of scope for the actual modeling language, but is understood on a less formal level and conveys important clarifying details. An example for the latter would be similar spatial arrangement of model elements even in a connection-based language to stress the point that structures are similar, if not related.

The authors argue that for supporting relaxed conformance in a (graphical) editor, there must be support of model transformations on the "relaxed" concrete syntax to transform it back to fully compliant concrete syntax, which can then in turn be correctly transformed to abstract syntax.

Relevance This paper is relevant as it outlines some of the important functionality and design goals to be implemented in this thesis – such as the possibility to leave edge ends open instead of being forced to connect them to another element, or the possibility to execute model transformations (i.e. graphical quick fixes) to a malformed graphical model to make it adhere to a defined metamodel.

Delimitation The paper provides the aforementioned ideas, but omits the actual execution of the concepts proposed. The prototypical realization of some of these ideas should be explored in this thesis.

4.3 Model editor usability

In constructing the prototype, an eye should be kept on the issues users frequently have with existing graphical editors and it should be considered how to avoid them. There are some existing articles on the subject of graphical model editor usability, as well as the general problems faced in MDE adoption, although none found so far touch on whether or not strict / implicit conformance enforcement is a source of frustration for users.

4.3.1 J. Rouly, J. Orbeck et al., Usability and Suitability Survey of Features in Visual IDEs for Non-Programmers [16]

Summary The authors of this paper have selected and investigated 25 commercial and open-source software for graphical modeling not only from general spoftware development notations (e.g. UML tools), but also from other domains working on graphic representations such as 3D modeling or DSL use-cases like radio circuitry development.

First, an overview of the considered properties and features of visual IDEs is given. The paper then evaluates the collected tools according to these criteria, based on to static perception of the user interface and the ease of use concerning certain example tasks. Furthermore, a comparison of the IDEs is made according to features offered by them as well as if and how certain interaction aspects, such as model correctness checking or element creation are implemented. Apart from that, a short description of the tools and arguments for the ratings are given.

Relevance This work was deemed relevant for the prototype development for two reasons: First, it provides an overview of possible visual IDE features and design decisions, which can be used as a starting point for requirement and interaction scenario definition. For example, the *Mode of Element Creation* is considered, which determines how the user may interact with the user interface to insert an element to the editing canvas: In some tools, this is accomplished by dragging elements

into the canvas, in others the user first selects a "painting tool", then clicks into the canvas to insert an element of the selected kind.

On the other hand, a selection of tools is given which, combined with the criteria above, can be investigated as a source for existing tools similar to the proposed relaxed conformance editor. The relevant criteria values are as follows: The *Domain* would be "Software", *Output* is "Direct Live", which means the user interacts directly with the graphical representation and the output reacts immediately. *Syntax enforcement* is "explicit", i.e. the software will show errors in the created models to the user instead of implicitly disallowing anything non-conformant. According to these criteria, only the tool *VioletUML* fits, which is a very simple drawing program for a limited number of UML diagrams. This tool, in spite of being marked with "explicit" syntax check, does not actually seem to possess such a feature; for example, circular inheritance relations or an inheritance relation from a class to a package are not detected in any way. As a conclusion, there is no tool in the list which particularly fits the targets of the planned prototype system.

4.3.2 P. Pourali, J. M. Atlee, An Empirical Investigation to Understand the Difficulties and Challenges of Software Modellers When Using Modelling Tools [17]

Summary This study again evaluates a selection of eight UML modeling tools, this time by having test subjects execute a number of different tasks. The metrics considered here are the effectiveness (number of errors during task execution) and efficiency (time taken for a certain task) of the tools. Furthermore, by means of a questionnaire it was assessed to which extent the users' *expectations* regarding the tools were met and how high the level of *satisfaction* was.

As the most pressing challenges in regard to modeling tasks, the areas of *Context* (that means keeping and remembering the link between different diagrams or different parts of one diagram) and *Debugging* (detecting and solving mistakes such as inconsistencies between different diagrams or invalid syntax constructs) were found. The authors provide some suggestions on how to solve these problems: One feature they would deem very helpful is the addition of automatic and real-time syntax checking, as test subject frequently did not make use of manual variants of these features in spite of them being present.

Relevance Like [16] described above, this study is primarily relevant because it identifies which features are commonly missing or not ideal in currently available modeling tools, and consequently some of these ideas can be incorporated in the prototype system. For example, the problem of having to manually invoke the error-checking and the difficulty of finding solutions for the errors detected would be solved by the *suggestions* feature as described in section 7.

Furthermore, in the background section, the authors name some tools with model assist features in place. Looking at these tools, the *YAKINDU Statechart Tools* is a good example of the implementation of instant consistency checking: Model errors (such as states without incoming connections) are marked directly in the model. This can serve as input for the user interface of the prototype to be developed.

5 Research questions

RQ1 What could the implementation and architecture of a graphical model editing tool, which supports free editing of the diagram without strictly enforcing conformance, but still enables automatic processing of the semantical meaning of the models, look like?

RQ2 Is the proposed architecture suitable for detection of typical scenarios in model driven engineering?

6 The Design - Methods and Procedures

As the main goal of the project is developing a prototype of a graphical model editing tool, the *Design (Science) Research* (DSR) approach is the central paradigm used. According to the methodology presented in [18], a design science research process consists of five to six steps, which will be outlined in the following sections along with their specific realization in the proposed project.

These activities strive to answer the research questions presented in 5. In what depth and thoroughness each of the questions will be addressed is not completely clear yet, as the available time depends heavily on the effort required for the others – especially the implementation of the prototype.

6.1 Problem Identification and Motivation

The first activity in the DSR process consists of the definition of a specific problem to be solved and in justification of the value of the solution. The problem can be further split up to aid in reasoning [18].

Broadly spoken, the problem to be addressed through this project is the improvement of usability in visual model editing tools, the problems of which can be partially found in previous studies (see section 4.3). The more specific sub-problems to be considered are the implementation of less strictly defined concrete syntax (i.e. relaxed conformance enforcement), for which the actual benefit is yet to be evaluated, and the context-aware error recognition / action suggestion as also proposed in [17].

The reasoning for the value of the solution was given in section 1 – with visual editing tools being a key workflow part in many applications across multiple domains, it seems worthwhile to investigate approaches that may increase their usability and suitability to common process models.

6.2 Definition of solution objectives

After the problem has been defined, the key aspects to be achieved by the proposed solution should be outlined ([18]).

The basic objectives of the project are as stated in 3: Provide a concept and architecture for flexible editing in which the definition of abstract and concrete syntax is decoupled.

As a prototypical implementation of the concept, an editor will be implemented. The language used as a case study will be a subset of the widely used state chart notation (see e.g. [19]), combining properties of connection-based and containment-based visual languages (see 4.1.2).

The following list shows the most important examples of notation elements in state-charts, and explains their relevance:

- *States*, represented by rounded rectangles with optional labels anywhere in their interior, or rounded rectangles split into two compartments by a solid line, with a label residing in the upper compartment
- *Initial states*, represented by their standardized icons
- *Transitions*, represented by segmented lines with labels representing their associated events. These serve as an example for a connection-based mechanism in a modeling notation.
- *Composite states*, represented by a "state" representation containing other model elements. This is an example of a containment-based mechanism, which means that state charts are not strictly graph-/connection based.
- *Orthogonal states / Parallel states*, represented by a composite state that is split into two (potentially more) compartments by a dashed line. This, along with the placement of labels in the upper "compartments" of states, is am example for a special relation of elements in a graphical notation:

The parent state is *split* into two child states with special meaning (i.e. parallelism) by an element included in it.

As stated before, the application should check the conformance of the model to a set of criteria, especially those that hinder correct interpretation of the diagram. Users should be able to freely modify the diagram, then get a hint for the incorrectness by the user interface and, if possible, be offered an action to rectify it.

Examples of constraints to be checked include:

- States must not overlap, except one is a child of the other; in this case they must be fully inside the parent's representation
- Transitions must be connected to a state on both ends
- The state chart itself and each sub-state may only have one initial state
- Transitions may not cross the lines separating the components of parallel states.

When there are multiple possible ways to interpret a graphical situation (such as a line being adjacent to multiple possible connection elements), the strategy of the system should depend on the correctness/plausibility of these interpretations - if only one interpretation strategy yields correct results, the other one should be automatically rejected. In the end, the system should provide an abstract syntax model of a drawing consisting of these elements, i.e. objects being instances of classes like "State" with attributes set according to the concrete information given in the diagram.

The flexibility on this stage will not include the relation of concrete to abstract syntax, for example defining notations different from those supported by the application will not be possible.

The most important non-functional characteristics will be the design for modular re-usability in respect to different languages and notations as well as the future extensibility for more features such as collaborative modeling [7].

6.3 Design and development

At the core of the DSR process, an artifact has to be designed and then developed. Artifacts can be contributions to theory or instances of theoretical constructs [18].

The first artifact to be developed is the architecture facilitating decoupled, relaxed conformance editing. As this architecture should be not only suitable for accommodating the exemplary concrete syntax metamodel, to this end multiple notations with widespread usage (for example the aforementioned state charts, UML sequence diagrams, activity diagrams) will be investigated for their characteristics. Example cases are demonstrated "on paper", like outlined in the listing of state-chart notation elements in section 6.2. Working out a full formal taxonomy of modeling languages is not the goal of this step, rather it is meant to demonstrate that the approach is applicable for multiple graphical notations. Based on the results of this inspection, definitions of architecture properties and system components needed will be worked out and documented.

Consequently, the other artifact will be an executable prototype of a software system usable for editing a graphical representation of a simple, limited modeling language (as explained in 6.2), as well as the code documentation of this system. The elaborateness and completeness of the implementation is not clearly defined yet, as the effort needed to implement both a sufficiently reliable, tested base system and the components needed for handling the state-chart notation is not entirely capable of being estimated in advance.

6.4 Demonstration and/or evaluation

The purpose of these two activities is checking whether or not the developed artifacts and their concepts solve the identified problems as assumed at the start of the project. While *demonstration* only demands a proof-of-concept, where the artifact is applied to at least one instance of the problem successfully, *evaluation* calls for a more rigid empiric or mathematical proof of the solution's ability to solve the given problem [18].

In this project, the goal of the evaluation is to gauge the usefulness and applicability both of the developed approach and its current state of implementation.

The suitability of the architecture design will be investigated as part of the prototype design phase – important cases commonly found in modeling notations, especially in the state chart notation used as the primary example, are identified and strategies for their handling (with the common components needed) are worked out. The evaluation of the prototype "in practice" will be conducted along a set of systematically developed test cases representing fragments of typical modeling situations, both in unit tests for single components and in functional tests of the complete system.

6.5 Communication

An important part of research is the communication and documentation of knowledge for others to build upon, most prominently in form of publications [18].

The communication part in this project will be the actual written form of the thesis, as well as the source code and documentation of the prototype for further use and development.

7 Ideas - Concepts and artifacts developed so far

This section contains part of the current state of the application concept. This does not have to be identical to the solution implemented in the final prototype, but is meant to give insights about the rough outline of the system in development.

7.1 Central technologies

For maximum platform independence along with a big selection of external tools and libraries, the application will be based on the Java Virtual Machine (JVM) platform, with a minimum requirement of version 1.8. However, instead of using Java directly, the Kotlin programming language[1] along with its JVM bindings will be used for its benefits in reducing boilerplate code and avoiding frequent sources of bugs by concepts like language-level null safety. As Kotlin provides seamless interoperability with Java code, libraries can be used nonetheless and without modification.

Another important technology will be the ReactiveX[2] paradigm, implemented by the libraries RxJava[3] and RxKotlin[4]. These libraries provide a powerful way of handling and manipulating event-driven data throughout an application, while also providing safe asynchronous processing facilities.

For the frontend component, due to the project time being too short for implementing rendering, dragging and other graphical editing aspects from scratch, the central technology used will be the Eclipse Graphical Editing Framework (GEF5)[5]. As this is a plain Java library, it can be used without dealing with the complexity of the Eclipse Rich Client Platform, instead the prototype will use a simpler User Interface based on JavaFX.

[1] https://kotlinlang.org/
[2] http://reactivex.io/
[3] https://github.com/ReactiveX/RxJava
[4] https://github.com/ReactiveX/RxKotlin
[5] https://www.eclipse.org/gef/

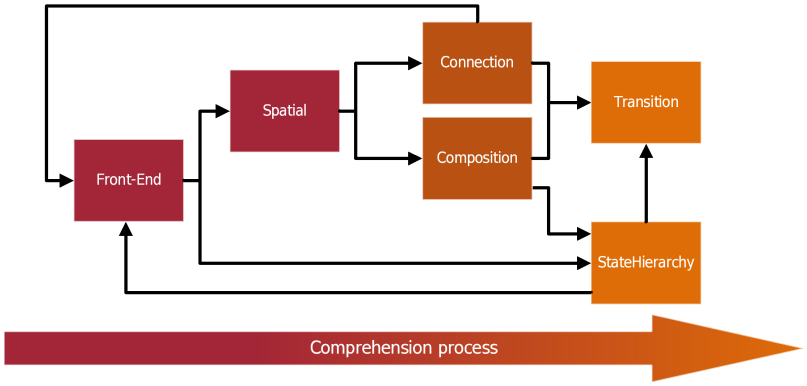

Figure 2: Example dependency network of Processors for state charts

7.2 Application architecture draft

Based on this foundation, a modular architecture was developed. In this architecture, true to the idea proposed in [6], the frontend component is decoupled from the abstract syntax aware part of the application, which is split up in multiple different *processors*. Each processor is an autonomous component, which should share no service instances with other parts of the application, and is designed to accomplish a single task – for example, one processor could be responsible for taking in changes in graphic objects as produced by the frontend, and calculating spatial relations (include, right-of, ...) between those, thus providing reusable, testable transformation steps for common properties of modeling languages. Processors store their data model through plain objects stored in a *repository*, which is exclusively manipulated by the processor itself to avoid concurrency problems. The repository stores a graph structure of model elements and their relations, which are specific to the processor at hand.

To illustrate the way processors might work together, a hypothetical structure of processors for the *comprehension process* of a state chart, i.e. the conversion from a concrete syntax instance to an abstract syntax instance (see also section 4.1.1]) and their interactions can be seen in figure 2. This structure only serves as an illustration; the processors needed in reality are to be determined during the first, conceptual phase of the project (see section 6.3]). It shows how basic graphical information from the frontend (graphic primitives and their locations) are abstracted step-by-step, for example regarding their spatial relations (right-of, include, ...) as these usually are more important to the concrete syntax than absolute coordinates.

The only connections processors have to the rest of the system are so-called *ModificationPorts*, through which all model operations occurring in the system will be propagated – processors receive *DiffCollections* representing atomic changes in the application model, and in turn produce another DiffCollection to be consumed by other processors. To continue the example above, another processor in a connection-based editing environment would consume the differences in spatial relations and mark possible connection ends whenever a line endpoint is nearby another graphic object, but not yet explicitly connected.

Figure 3 shows the execution procedure of a processor whenever a DiffCollection comes in.

All of the system facilities are encapsulated behind such processor instances and tied together by a central ModificationBusManager component tasked with connecting ModificationPort inputs and outputs to a central exchange.

Figure 4 shows a simplified exemplary configuration of the system, containing two processors: One is responsible for calculating spatial relations, the other one will suggest connection creation based on these relations. As can be seen from this diagram, the frontend does not differ in its connection to the ModificationBusManager from the “other” processors, but also works by consuming and outputting

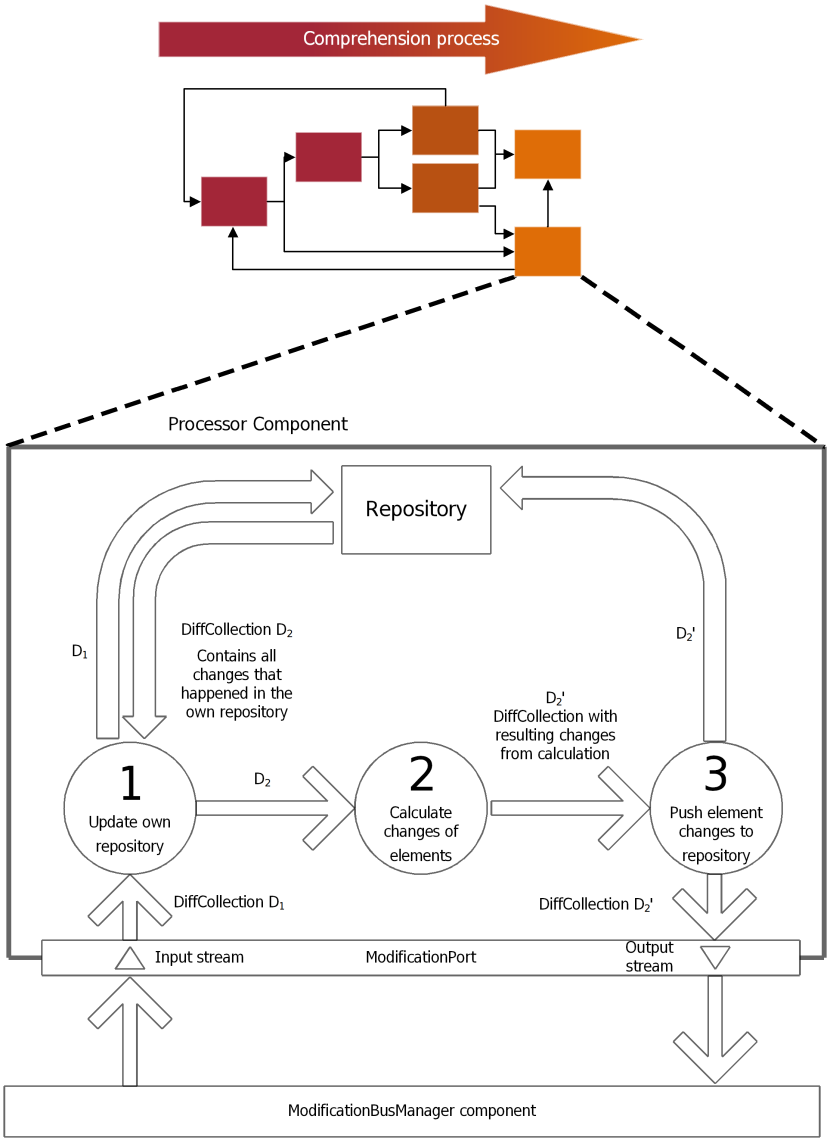

Figure 3: Schematic processing routine

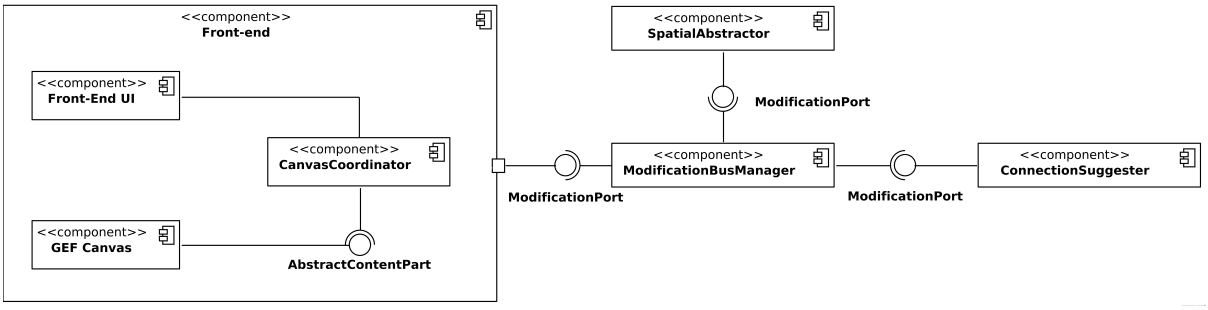

Figure 4: High-level component diagram of the application

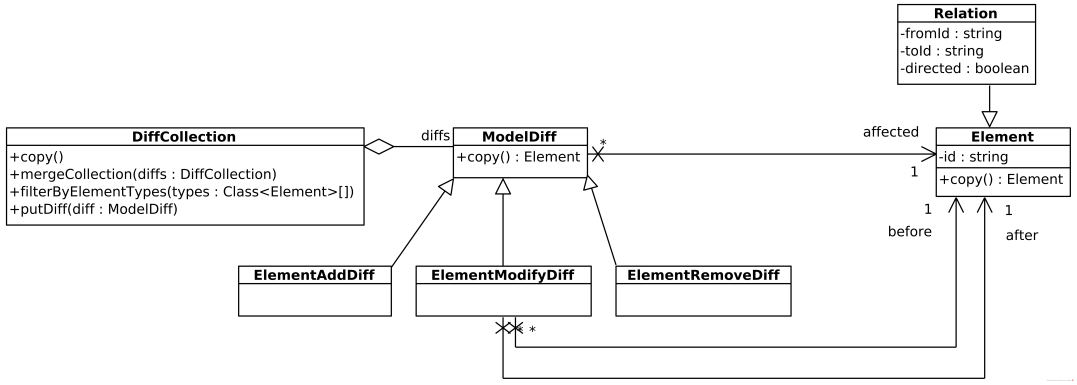

Figure 5: Basic entities in the application

model change events over reactive streams.

The basic entity classes in the application are shown in figure 5. The *DiffCollection* serves as the most fundamental unit of exchange. This is what a single event on the reactive facilities contains. DiffCollections can contain multiple *ModelDiff* instances, representing adding, modifying or deleting an element. The Element class is the common base class for processable entities, regardless of if they are belonging to the concrete or the abstract syntax domain. DiffCollections can be filtered by element types or other arbitrary predicates, which is done in order to ensure that no processor gets too much input that is not relevant for its task.

DiffCollection and Element objects are simple trees of data transfer objects without any dependencies on external services. Because of that, they can be duplicated by their copy() method and transferred between processors without shared instances. That way, all processors can run in parallel threads without the need for inter-processor locking and similar constructs.

7.3 Suggestion mechanism

One of the special features of the application will be the *suggestion* capability. This provides the link to abstract syntax operations that cannot (unambiguously) be expressed in concrete syntax concepts. For example, as stated before, connections will not have to be explicitly created, providing fixed start and end objects, rather they can also be created with only one end connected or completely free-floating. In some situations, it cannot be determined whether the user intended to create a connection to an element or if the line just ends there randomly. Because the front-end cannot actually "understand" whether

elements can be related as it is decoupled from the abstract syntax metamodel, a back-end processor must generate a suggestion in this case. The suggestion causes the frontend to display an action to the user which can then be clicked and executed to add the appropriate "transition" abstract syntax element, without the front-end actually having implemented the concept of transitions.

8 Limitations and Delimitations

Limitations are especially due to the time constraints of the project paired with the "vertical" goals, i.e. the goal to create a fully functional prototype. Most probably, it will not be possible to develop a fully featured editing application, nor will it be possible to develop a complete and correct syntax interpreter for state charts. The amount of features realized and contained in the result is very dependent on the effort needed for the groundwork in both the theoretical and implementation aspects.

For the conceptual phase, the priority will lie in showing the applicability of the concept to various modeling languages, not a complete formal taxonomy of notations or a complete listing of their properties.

In the development of the prototype, the emphasis will be put on the basic architecture and the construction of the frontend to achieve the decoupling explained in [6]. To showcase the ideas, an editor will be implemented for one simple graphical notation (see 6.2). The resulting system will be a prototype and thus be limited in performance, stability and user interface polish, although it will be tried to achieve all of this as much as possible.

What will also not be the subject of this project is formal definition of the abstract syntax metamodel, instead the corresponding processors will be implemented specifically for the chosen notation. Thus, it will not be possible to use EMF metamodels or other kinds of automatic editor generation / configuration, which could be the subject of future work based on the implemented prototype.

9 Significance of the study

This project will extend past research in relaxed model conformance, as outlined in section 4.2. There already are multiple different approaches to flexible design of metamodels, but either user interaction with a graphical notation is not considered at all, even if one could imagine support for this being helpful in the process of model creation [5], or the suppport for modeling notations remains on a "sketch" level that cannot be used for later, more strict phases of the process.

The aim is to fill that gap and provide insights about graphical editor design and architecture in relaxed conformance modeling, as well as the groundwork for further developments such as integrating standard metamodel definitions.

10 Planning

10.1 Own Background

I have general experience in software architecture and implementation from previous projects both in university and in industry, employing technologies like RxJava and Kotlin that are to be used in the thesis. I have also applied the Design Science Research methodology and evaluated an application by means of semi-structured interviews for my Bachelor's Thesis before. For the conceptual part, the most important knowledge to be gathered is that of visual languages and the way in which they express On the implementation side, what needs to be learned in the thesis is primarily working with GEF and the mitigating the challenges associated with the distributed processing model, as well as discovering the algorithms and libraries that are needed for the individual parts of the application.

10.2 Work packages

M1 Preparation of basics

- Get a grasp on the basic concepts of visual languages as outlined in 4.1.2, design data model on this basis
- Implementation of first proof-of-concept prototype with support for basic spatial relations and connections, first rudimentary "suggestion" support

M2 Finish basic architecture / develop concepts for language support

- As stated in section 6.3, analyze common properties / patterns in modeling languages
- As a basis for further development, introduce "debug" view into prototype to be able to properly see interpretation step results
- Test and improve / stabilize core components of prototype

M3 Design of architecture for state-charts

- Identify specific edge cases in modeling language types
- Based on these common properties, identify needed components to support those languages and their specifications
- Specifically, define which of these components will be needed for state-chart interpretation

M4 Begin with implementation of components needed for state-chart interpretation and identified in M3

M5 Evaluation / Improvement of resulting system

- Implement test cases as identified in M3
- Stabilize components from M5

M6 Write / finish written thesis

10.3 Contingency plan

The main risks in this projects lie within the maturity of the prototype required to effectively conduct an evaluation (due to no really suitable third-party applications being available), which has to be developed in a very short time frame considering the other activities required, making it necessary to rely on third-party frameworks for essential parts of the UI (especially GEF, the suitability of which is not entirely clear). Before the prototype is actually finished with the minimum requirements, it is rather unclear into which delays the project can run.

1. **Risk** Too complex / high effort to implement the planned feature set

   Contingency If not all features of the target notation can be implemented in time, some may have to be left out. As the basic realizability of these features is demonstrated during the conceptual phase, this can be taken as a basis for further development.

2. **Risk** Implementation performance is too low

   Contingency It could be that the original design idea needs to be "weakened", increasing the front-end's knowledge about the structure of the modeling language beyond just graphic primitives. The classes of modeling languages (see 4.1.2) could help find suitable abstractions. Furthermore, as the suitability of the approach is evaluated based on automatic tests, low front-end performance does not prevent showing the basic applicability of the concept. Rather possible points for performance improvements can be evaluated and implemented in future work.

References

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006, ISSN: 0018-9162. DOI: 10.1109/MC.2006.58. [Online]. Available: https://doi.org/10.1109/MC.2006.58.

[2] J. Davies, J. Gibbons, J. Welch, and E. Crichton, "Model-driven engineering of information systems: 10 years and 1000 versions," *Science of Computer Programming*, vol. 89, pp. 88–104, 2014, ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.02.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642313000270.

[3] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. 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*, ser. MODELS '16, New York, NY, USA: ACM, 2016, pp. 24–34, ISBN: 978-1-4503-4321-3. DOI: 10.1145/2976767.2976807. [Online]. Available: http://doi.acm.org/10.1145/2976767.2976807.

[4] J. Kärnä, J.-P. Tolvanen, and S. Kelly, "Evaluating the use of domain-specific modeling in practice," in *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM '09)*, M. Rossi, Ed., ser. Helsingin kauppakorkeakoulun julkaisuja, Helsinki: Helsingin Kauppakorkeakoulu, 2009, ISBN: 978-952-488-372-9. [Online]. Available: http://dsmforum.org/events/DSM09/Papers/Karna.pdf (visited on 04/06/2019).

[5] E. Guerra and J. de Lara, "On the quest for flexible modelling," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '18, New York, NY, USA: ACM, 2018, pp. 23–33, ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239376. [Online]. Available: http://doi.acm.org/10.1145/3239372.3239376.

[6] Y. van Tendeloo, S. van Mierlo, B. Meyers, and H. Vangheluwe, "Concrete syntax: A multi-paradigm modelling approach," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2017*, B. Combemale, M. Mernik, and B. Rumpe, Eds., New York, New York, USA: ACM Press, 2017, pp. 182–193, ISBN: 9781450355254. DOI: 10.1145/3136014.3136017.

[7] Y. Van Tendeloo and H. Vangheluwe, "Unifying model- and screen sharing: 2018 ieee 27th international conference on enabling technologies: Infrastructure for collaborative enterprises (wetice)," in *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2018. DOI: 10.1109/WETICE.2018.00031.

[8] N. Hili, "A metamodeling framework for promoting flexibility and creativity over strict model conformance," in *Proceedings of the 2nd Workshop on Flexible Model Driven Engineering co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016)*, ser. FlexMDE@MoDELS, 2016. [Online]. Available: http://ceur-ws.org/Vol-1694/FlexMDE2016_paper_6.pdf.

[9] A. Hartman, D. Kreische, F. Fondement, and T. Baar, "Making metamodels aware of concrete syntax: Model driven architecture – foundations and applications," in *Lecture Notes in Computer Science, vol. 3748 // Model driven architecture*, ser. Lecture notes in computer science, A. Hartman and D. Kreische, Eds., vol. 3748, Berlin and New York: Springer Berlin Heidelberg and Springer, 2005, pp. 190–204, ISBN: 978-3-540-32093-7. DOI: 10.1007/11581741{\textunderscore}15.

[10] A. Kleppe, "A language description is more than a metamodel," in *Fourth International Workshop on Software Language Engineering*, megaplanet.org, 2007, pp. –.

[11] T. Baar, "Correctly defined concrete syntax," *Software & Systems Modeling*, vol. 7, no. 4, pp. 383–398, 2008, ISSN: 1619-1374. DOI: 10.1007/s10270-008-0086-z. [Online]. Available: https://doi.org/10.1007/s10270-008-0086-z.

[12] P. Bottoni and G. Costagliola, "On the definition of visual languages and their editors: Diagrammatic representation and inference," in *Diagrammatic Representation and Inference*, M. Hegarty, B. Meyer, and N. H. Narayanan, Eds., ser. Lecture notes in computer science, Berlin and Heidelberg: Springer, 2002, ISBN: 978-3-540-46037-4.

[13] J.-S. Sottet and N. Biri, "Jsmf: A flexible javascript modelling framework," in *Proceedings of the 2nd Workshop on Flexible Model Driven Engineering co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016)*, ser. FlexMDE@MoDELS, 2016. [Online]. Available: `http://ceur-ws.org/Vol-1694/FlexMDE2016_paper_5.pdf`.

[14] G. Costagliola, A. De Lucia, S. Orefice, and G. Polese, "A classification framework to support the design of visual languages," *Journal of Visual Languages & Computing*, vol. 13, no. 6, pp. 573–600, 2002, ISSN: 1045926X. DOI: `10.1006/jvlc.2002.0234`.

[15] R. Salay and M. Chechik, "Supporting agility in mde through modeling language relaxation," in *Proceedings of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2013), Miami, Florida, USA, September 29, 2013*, J. de Lara, D. Di Ruscio, and A. Pierantonio, Eds., ser. CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 20–27. [Online]. Available: `http://ceur-ws.org/Vol-1089/3.pdf`.

[16] J. M. Rouly, J. D. Orbeck, and E. Syriani, "Usability and suitability survey of features in visual ides for non-programmers," in *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '14, New York, NY, USA: ACM, 2014, pp. 31–42, ISBN: 978-1-4503-2277-5. DOI: `10.1145/2688204.2688207`. [Online]. Available: `http://doi.acm.org/10.1145/2688204.2688207`.

[17] P. Pourali and J. M. Atlee, "An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '18, New York, NY, USA: ACM, 2018, pp. 224–234, ISBN: 978-1-4503-4949-9. DOI: `10.1145/3239372.3239400`. [Online]. Available: `http://doi.acm.org/10.1145/3239372.3239400`.

[18] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007. DOI: `10.2753/MIS0742-1222240302`.

[19] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987, ISSN: 0167-6423. DOI: `10.1016/0167-6423(87)90035-9`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/0167642387900359`.