

Toward Intelligent Generation of Tailored Graphical Concrete Syntax

Meriem Ben Chaaben*
meriem.ben.chaaben@umontreal.ca
Université de Montréal
Montréal, Canada

Oussama Ben Sghaier*
oussama.ben.sghaier@umontreal.ca
Université de Montréal
Montréal, Canada

Mouna Dhaouadi
mouna.dhaouadi@umontreal.ca
Université de Montréal
Montréal, Canada

Nafisa Elrasheed
nafisa.elrasheed@gmail.com
Polytechnique Montréal
Montréal, Canada

Ikram Darif
ikram.darif.1@ens.etsmtl.ca
École de Technologie Supérieure
Montréal, Canada

Imen Jaoua
imen.jaoua@umontreal.ca
Université de Montréal
Montréal, Canada

Bentley Oakes
bentley.oakes@polymtl.ca
Polytechnique Montréal
Montréal, Canada

Eugene Syriani
syriani@iro.umontreal.ca
Université de Montréal
Montréal, Canada

Mohammad Hamdaqa
mhamdaqa@polymtl.ca
Polytechnique Montréal
Montréal, Canada

ABSTRACT

In model-driven engineering, the concrete syntax of a domain-specific modeling language (DSML) is fundamental as it constitutes the primary point of interaction between the user and the DSML. Nevertheless, the conventional one-size-fits-all approach to concrete syntax often undermines the effectiveness of DSMLs, as it fails to accommodate the diverse constraints and specific requirements inherent to diverse users and usage contexts. Such shortcomings can lead to a significant decline in the performance, usability, and efficiency of DSMLs. This vision paper proposes a conceptual framework to generate concrete syntax intelligently. Our framework considers multiple concerns of users and aims to align the concrete syntax with the context of the DSML usage. Additionally, we detail a baseline process to employ our framework in practice, leveraging large language models to expedite the generation of tailored concrete syntax. We illustrate the potential of our vision with two concrete examples and discuss the shortcomings and research challenges of current intelligent generation techniques.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages; Specialized application languages; Visual languages; Graphical user interface languages.**

* Both authors contributed equally to the paper.

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

MODELS '24, September 22–27, 2024, Linz, Austria

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0504-5/24/09

<https://doi.org/10.1145/3640310.3674085>

KEYWORDS

Domain-specific Modeling Languages, Concrete Syntax, Artificial Intelligence, Large Language Models.

ACM Reference Format:

Meriem Ben Chaaben*, Oussama Ben Sghaier*, Mouna Dhaouadi, Nafisa Elrasheed, Ikram Darif, Imen Jaoua, Bentley Oakes, Eugene Syriani, and Mohammad Hamdaqa. 2024. Toward Intelligent Generation of Tailored Graphical Concrete Syntax. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24)*, September 22–27, 2024, Linz, Austria. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3640310.3674085>

1 INTRODUCTION

Domain-specific modeling languages (DSMLs) are high-level and specialized languages tailored to a particular application domain [28]. They enable domain experts, who are not necessarily programmers, to design solutions to problems related to a specific domain, often leveraging model-driven engineering (MDE) techniques and tools. As they are closer to the problem domain than general-purpose programming languages, their custom nature plays an increasingly important role in multiple areas of software engineering [2, 48]. DSMLs are primarily engineered with an abstract and a concrete syntax [22], where the abstract syntax describes the conceptual elements of the language, usually through a metamodel [48]. The concrete syntax (CS) provides a visual representation (e.g., textual, graphical, tabular, etc.) of the conceptual elements defined within the abstract syntax [17].

The CS bridges the semantic gap between abstract models and their real-world applications by providing the means to express domain-specific concepts understandably and intuitively. This enables domain experts to interact with models effectively, paving the way for more efficient and reliable software development. The CS is also the first contact that domain experts have with the DSML, thus it plays a crucial role in both the acceptance of the DSML by domain experts and the definition of notations that meet their expectations. However, despite the significant impact that the CS has on the

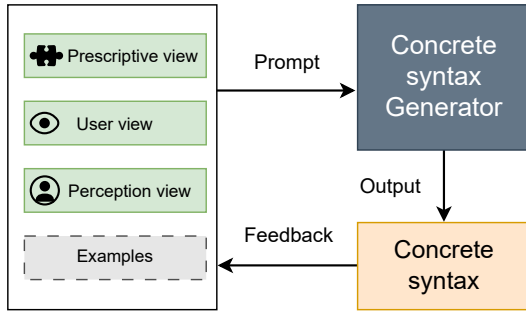


Figure 1: Overview of our proposed framework for the intelligent generation of tailored concrete syntax

quality of the DSMLs, current DSML development practices mainly focus on the abstract syntax and treat the CS as a byproduct [64].

Since CS serves as the primary interface for users, it is crucial to prioritize clarity, intuitiveness, and efficiency in its design [39]. The DSML engineer needs to possess a multitude of artistic, design, user experience, technical, and prior knowledge skills to create an effective CS, thus creating a significant usage barrier. Moreover, these challenges are further exacerbated by using mixed or inadequately adapted notations, which can significantly complicate user interactions. Such inconsistencies in notations can lead to confusion and reduce the effectiveness of the CS in conveying the intended meaning of the model [42].

The subjective nature of CS introduces additional complexities, such as user preferences, profiles, and context variability. Consequently, a one-size-fits-all approach to CS may prove ineffective, as it might not meet all users' diverse needs and preferences, potentially hindering overall effectiveness and user experience. However, creating a customized CS is both time-consuming and labor-intensive. Thus, there is a need for an automated process for the creation of tailored CS.

Previous research has attempted to generate CS automatically. Nastov and Pfister [43] proposed a semi-automated CS generator using a graphical role election process. Muller et al. [41] proposed to generate CS from textual templates and UML class diagrams. Other research tried to customize CS to incorporate different user profiles, e.g., to account for the visually-impaired users when manipulating UML diagrams [35], or when constructing relational diagrams by proposing an audio interface [38]. To the best of our knowledge, no previous work has attempted to investigate the possibility of automatically generating customized CS.

Large Language Models (LLMs) have shown promising potential in automating different MDE tasks, such as model completion and metamodel recommenders [9, 11]. In our vision of automatically generating tailored CS for DSMLs, we investigate if an LLM-based solution is viable. We focus specifically on graphical CS. Our contribution is **a conceptual framework designed to intelligently generate tailored graphical CS**. As depicted in Figure 1, the framework accounts for multiple views, namely the perspective view, the perception view, and the user view, to take into account different concerns, including the preferences of the domain experts. We envision using LLMs to iteratively generate customized graphical CS based on prompt engineering approaches guided by diverse

feedback signals, including user and tool feedback. Here, tool feedback could be quantitative or qualitative measurements of the CS. We instantiate our vision on two DSMLs: a simple 'mind map' language and a 'traffic light' language with more complex context and preferences. The experiments of this paper (i.e., conversations with ChatGPT and metamodel representations) are provided in the supplementary material.

The remainder of this paper is structured as follows. Section 2 outlines related works. Section 3 details our proposed framework. Section 4 elaborates on our baseline framework application process on a mind-map example. Section 5 discusses the benefits, limitations, and research challenges of our proposed framework based on a traffic light DSML. Section 6 concludes with future work.

2 RELATED WORK

This section examines current methodologies for defining the graphical CS of DSMLs and discusses their limitations. We also investigate the benefits of integrating user constraints to ensure practical alignment. Lastly, we explore the impact of artificial intelligence, specifically LLMs, on DSML development.

2.1 Defining Graphical Concrete Syntax

DSMLs employ graphical, textual, or mixed syntax to model specific domains. However, current DSML development practices often prioritize the abstract syntax over the CS [64]. Fondement and Baar [18] formally specify the graphical representation of language concepts, extending the abstract syntax metamodel with visual descriptions and emphasizing Object Constraint Language (OCL) for constraints. Baar [5] introduce a standardized CS format, enabling diverse model expressions through display manager classes that bridge the abstract syntax with visual and textual representations.

The graphical CS can be defined in several ways: (1) by explicitly mapping icons and splines to the abstract syntax, as done in tools like AToMPPM [55] and MetaEdit+ [25]; (2) by creating explicit models that link to metamodel elements and support complex queries, such as Sirius Viović et al. [59]; and (3) by incorporating annotations directly in the metamodel, as in Eugenia Kolovos et al. [27]. In this paper, we adopt the first method, explicitly mapping icons and splines to the abstract syntax elements.

Some approaches focus on the requirements engineering phase of the DSML development life-cycle. Cho et al. [12] propose that domain experts specify first-class graphical DSML requirements through graphical notations and generate the metamodel from it. Pescador and de Lara [44] propose a mind map approach to illustrate DSML requirements. A central concept symbolizes the DSML and branches into three key elements: structure, behavior, and organization. This approach facilitates model creation, detailed in [1], using flexible transformations from a metamodel design pattern collection. Van Tendeloo et al. [58] tackles DSML limitations by explicitly modeling bidirectional mappings between abstract and CS. This "perceptualization" approach clearly separates the back-end and front-end of modeling tools and provides flexibility for supporting multiple front-ends and diverse representations.

2.2 Capturing User Constraints

Software requirement specifications are crucial for communicating the stakeholders' needs. They are usually expressed in natural language. Deeptimahanti and Sanyal [14] use natural language processing techniques such as parsing, pronoun resolution, and morphological analysis to bridge the gap between requirement specifications and graphical CS, and streamline the translation of natural language requirements into UML models for comprehensive software design.

For the customization of software to meet end-user needs, a literature review [6] categorized related research into three topics: end-user development, end-user programming, and end-user software engineering. The survey explores techniques like component-based customization, programming by example, and digital sketching. While focusing on customization, the survey does not specify whether the collected research allows intelligently generated CS or relies solely on manual user specification.

Considering user preferences in software customization is crucial. Li et al. [34] propose using LLMs to capture human preferences through prompting, active learning, or interactive questions (generative elicitation). Costa et al. [13] incorporate ontologies to capture accessibility information, such as the impairment or disability of users and modifications to software or hardware to improve user interactions [24, 37].

2.3 Generation of Language Aspects

There has been a significant recent shift towards employing AI-based methodologies to (semi)-automate MDE modeling tasks.

2.3.1 LLMs for language engineering Weyssow et al. [62] introduce a deep learning-based approach, trained on numerous metamodels, to assist in domain concept recommendation during metamodeling. Other works target model synthesis. For instance, Rahimi et al. [45] use Generative Adversarial Networks to create realistic model instances. Yang and Sahraoui [65] process natural language specifications using machine learning to build complete models.

Prompting strategies play an essential role in guiding the LLMs. These strategies include: (i) instruction-based prompting that involves giving clear instructions to the learning model [66], (ii) few-shot prompting that involves guiding the learning model in performing tasks using some examples [52], and (iii) chain of thoughts prompting that involves guiding an exploration of ideas through a series of connected or logically sequential questions and prompts to refine the understanding of the learning model [61].

Arulmohan et al. [4] employ prompt engineering through a chain of prompts by extracting the concepts from natural language specifications, categorizing them, and then identifying relationships to synthesize models. Alternatively, Hans-Georg et al. [21] propose to synthesize models by prompt engineering GPT-4 to generate models given the specifications directly. Chaaben et al. [9] uses few-shot prompt learning for model completion tasks where the partial model is provided, and GPT-3 is prompted to suggest related elements. Furthermore, Tinnes et al. [56] propose fine-tuning GPT-3 on edit patterns for models to generate recommendations during the evolution process of the model. In addition to model synthesis, LLMs

have been used to enrich automatic code generation (i.e., model-to-text transformations) by integrating the textual representation of the models and constraints directly into the prompt [50].

Overall, most studies employ LLMs for MDE tasks by either (i) guiding LLMs with specific prompts and defining desired outputs or (ii) fine-tuning the LLMs with a few-shot approach.

2.3.2 Integrating feedback for LLMs Incorporating feedback can refine output as it enhances the accuracy of models and facilitates the fine-tuning of the generation process.

End-user feedback provides valuable insights into the relevance of the generated output and its alignment with the users' expectations and preferences [15, 31]. *Deep learning model feedback* derives from feedback generated by another deep learning model designed to evaluate specific characteristics of the generated output [23] (e.g., a deep learning model that predicts if a generated metamodel conforms to requirements). Lastly, *tools feedback* encompasses feedback from auxiliary tools that evaluate the quality and correctness of the generated output [40] (e.g., analyzers, linters, and quality assessment tools).

2.3.3 Input representation for LLMs Besides feedback, the effectiveness of LLMs is significantly affected by the representation of the input data. Throughout the literature, some languages and tools were proposed to represent metamodels that are understandable by LLMs. PlantUML¹ is an open-source tool that allows users to create diagrams using a simple and intuitive textual language. It supports various types of diagrams (e.g., UML diagrams), and it can be understood by LLMs such as ChatGPT [21]. Other works created DSMLs that are engineered specifically for machine learning pipelines. For instance, Rajaei et al. [46] proposed a DSML for encoding models into valid input graphs for graph-learning tools.

2.4 Synthesis

In conclusion, the existing literature demonstrates considerable progress in generating graphical CS using various methodologies. However, a notable research gap remains in developing customized CS generation to accommodate user constraints. Most of the works on concrete syntax engineering focus on textual domain-specific modeling languages, whereas our vision aims to automate graphical concrete syntax generation. State-of-the-art works often overlook dimensions like user preferences, which our framework explicitly incorporates as information sources for LLMs. We target both language engineers, who design and customize the concrete syntax, and modelers, who seek to personalize it. Importantly, our framework does not aim to generate or create the model editor itself but to create graphical concrete syntax for individual language elements, assuming existing technical infrastructure for integration. Additionally, despite the current exploration of LLMs for MDE, which has demonstrated their effectiveness in various modeling tasks, their application in the specific area of CS generation remains largely unexplored.

3 VIEWS TO TAILOR CONCRETE SYNTAX

This section describes the main contribution of this article: a conceptual framework for intelligently generating graphical CS tailored for

¹<https://plantuml.com/en/>

a particular usage. Figure 2 depicts an overview of our conceptual framework. It comprises three main viewpoints, each providing unique considerations to integrate into the generation process. Note that specifying all viewpoints and their components is optional. However, providing more details to the intelligent generator will likely improve the desired generated output.

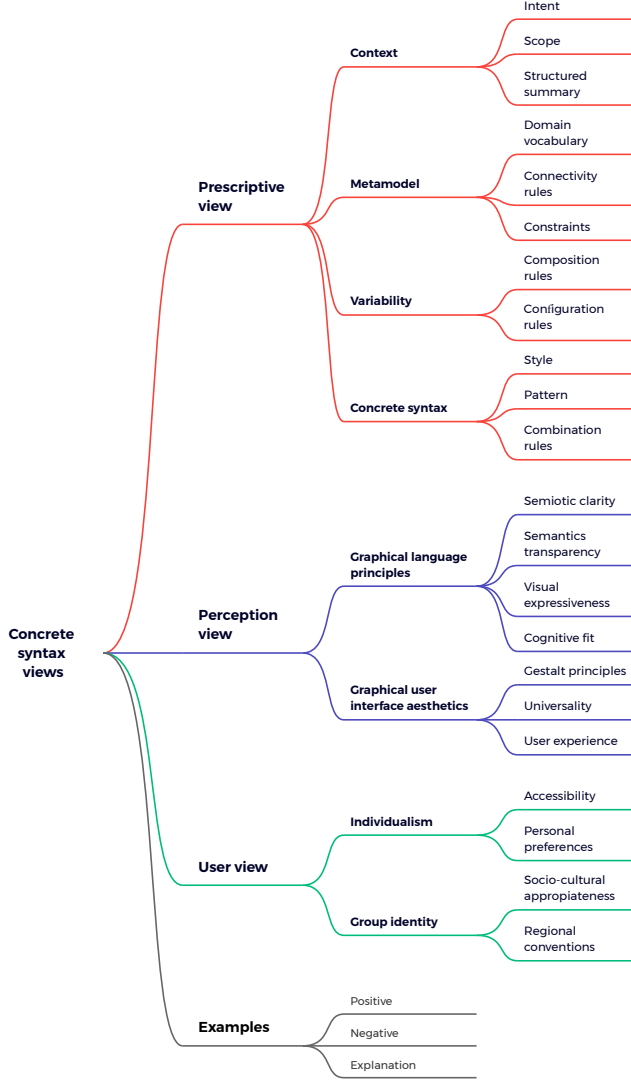


Figure 2: The proposed input viewpoints for the generation of tailored concrete syntax in our framework

We derived these views after a thorough literature analysis, experience reports of industrial case studies, and our experience in developing graphical domain-specific modeling languages. The views and subcategories were carefully selected based on established principles and standards in the field, ensuring a robust and comprehensive approach to graphical CS generation. Although this list may not be exhaustive, it outlines important dimensions to consider when engineering the graphical CS of a modeling language.

We detail and illustrate the proposed framework on the ‘mind map’ example. Recall that the mind map is a visual formalism that organizes information, ideas, or thoughts around a central concept. This central concept extends branches to represent related topics and subtopics, creating a hierarchical structure.

3.1 Prescriptive View

This view focuses on formally defining the syntax of the DSML by establishing essential guidelines and configurations that adapt to the specific needs of different situations and users. The language engineer defines this view to provide the necessary context for the LLM. This view consists mainly of four components. The **context** considers the *intent* of the entire language that defines the rationale behind it and its target users. The DSML *scope* determines what concepts are included or excluded from the DSML to restrict the design space only to elements relevant to this DSML. A *structured summary* describing the environment in which the CS will be used could also be provided. In the mind map DSML, we specify that it is intended for usage, such as supporting brainstorming activities during meetings involving several stakeholders.

The **metamodel** defines the abstract syntax of the language to specify a model [30]. More specifically, it details concepts and their attributes in a *domain vocabulary*. Concepts can be related to each other according to *connectivity rules* (e.g., dependency, ontology, composition). These elements can be further *constrained* to form a coherent structure that the CS shall render. For example, the metamodel for mind map defines concepts such as “Main Topic”, “Sub Topic”, and “Tag”.

Sometimes, distinct concepts share the same properties, e.g., the main topic and subtopics should have the same shape. In this case, one can specify the **variability** of concepts [57], through *composition rules* and *configuration rules* to allow custom arrangements and customizations to suit different variants of the same concept. For example, a topic could be represented differently if it has zero, one, or many sub-topics.

Some desired features of the CS may be imposed. One may enforce a certain representation *style* of the visualizations, such as a specific theme or palette of colors. There may be common *patterns* for repetitive concepts or *combination rules* of the CS element, e.g., preserving a certain distance between subtopics.

3.2 Perception View

This view adds improvements to the overall CS specification following design principles and aesthetics. It encompasses the common knowledge that the LLM is either finetuned on or has a knowledge base that it can access to get relevant design laws.

Graphical language principles for graphical CS, like those defined in [39] ensure that it is appropriate for the DSML users. In particular, *semiotic clarity* ensures that a CS representation is assigned to all the metamodel elements provided in the prescriptive view without redundancy. Adjusting the degree of *semantic transparency* enables the generation of visualizations that are more appealing to the user [29]. *Visual expressiveness* determines how to vary graphical notations along eight visual variables. Overall, the generated syntax representations should be a *cognitive fit* to the DSML users to improve their problem-solving performance [20, 51].

Graphical user interface principles advocate for a syntax that is not only functional but also cognitively and visually engaging to DSML users. For instance, Gestalt principles [63] provide general guidelines for organizing visual objects, which help gear the variability in the prescriptive view. Providing *universality* information helps the generator in choosing symbols that are understood by the target DSML users. Concerns related to improving the user interface design to optimize the DSML *user experience* can also be specified [19], e.g., to facilitate the rapid creation of sibling topics or sub-topics in a mind map.

3.3 User View

This view tailors the CS to user preferences. **Individualism** preferences are specific to a user who may express their *personal preferences* about, for instance, style and iconography. Users with *accessibility* issues can also specify this information to ensure that the CS is adapted for physical impairments with font size or color choice. For instance, specifying that the mind map should be adapted to color-blind users. Thus, the main topic must be distinguishable from the subtopics in both color and size.

Group identity preferences encompass collective preferences about the *regional conventions* and policies where the typical DSML users reside—for example, formatting phone numbers and addresses according to the country or representing trees as evergreen trees for a DSML targeted for Nordic conditions. When specified, the generated CS shall also be *socio-culturally appropriate* to be inclusive and diverse and avoid a syntax that may offend individuals. For example, an icon representing a user may use diverse skin colors or simply use a stick figure.

3.4 Positive and Negative Examples

At the confluence of these viewpoints stands the CS generator, an essential LLM-based component that integrates inputs from each viewpoint to create a CS that is both precise in its domain representation and customized to user specifications (c.f. Figure 1). To effectively guide this generation, instantiated **examples** can be provided for each element within the proposed framework. *Positive (negative)* examples demonstrate similar (un)desired CS characteristics. An *explanation* can be provided with examples to complement few-shot learning with a chain of thought.

For instance, sample figures can be provided, along with an explanation of how the symbol matches its usage, e.g., “these emoticons should be used as tags in the mind map”. It is also a means of specifying user preferences, e.g., “I like the styling of this line”. The generator would then consider this extra information when outputting the CS.

4 INTELLIGENT GENERATION FRAMEWORK FOR TAILORED CONCRETE SYNTAX

To systematically identify and illustrate the research gaps towards our vision, we provide a baseline process to apply our intelligent generation framework for tailored CS. Following state-of-the-art techniques in software engineering, we integrate LLMs into our process. The steps were determined by the authors through experimentation and are grounded in extensive previous research into applying LLMs to modeling tasks. The application detailed here

makes the intelligent generation tangible, highlights the research gaps in current methodologies and tools, and serves as a baseline for future research.

Specifically, we propose an iterative process designed to guide the user’s interactions with the LLM within the context of our framework. Figure 3 outlines the different steps of this process. The steps annotated with a user are performed outside the LLM-based generator, while the LLM performs the others. Note that the user steps can be wholly or partly automated. We designed this process to be highly iterative, as the LLM is unlikely to provide the desired output on the first attempt. Continuously refining the prompts has also been identified as an effective technique for improving the responses of the model [16].

Table 1 illustrates the proposed process on the mind map example using ChatGPT4. We specifically chose the learning model gpt-4-1106-vision-preview as it is a well-known LLM that supports image generation using the integrated DALL-E [47].

Our baseline process includes two main phases. First, we use the LLM to generate a textual representation of the CS. Then, we guide it into translating the generated text into a graphical representation. We introduce this intermediary textual phase based on insights from the state-of-practice [36] suggesting that, while LLMs are highly proficient in text generation and comprehension, their direct graphical representation capabilities can be less reliable, especially for complex and detail-rich graphics [49]. A key aspect of this baseline process is the prompt engineering technique. We report on our strategic formulation and iteration of inputs to elicit the most relevant and accurate outputs generated by the LLM.

4.1 Step 1: Metamodel Pre-processing

First, the user shall prepare the material to be used as input for the generation of the CS. Assuming the metamodel of the DSML is already available, the preprocessing step comprises several activities, including metamodel slicing and encoding. Slicing the metamodel allows for separating large metamodels into smaller, more manageable ones to avoid lengthy prompts. Furthermore, it enables the generator to focus on specific metamodel elements, which reduces the complexity of the metamodel and increases its understandability [7]. Slicing can be static, where the slicing operation does not interpret the model at hand, or dynamic, where parts of the metamodel are evaluated for the operation.

Encoding the metamodel accurately for processing is critical in generating tailored CS. This preprocessing step translates the metamodel into a format that is understandable by the used LLM. For instance, here we use the PlantUML syntax to convert the mind map metamodel into a textual format that the LLM understands, as presented in Listing 1. For the mind map example, since the abstract syntax metamodel is simple and only includes a few concepts, we did not need to perform slicing on the metamodel.

4.2 Step 2: Initial Prompt- Prescriptive view specification

Next, the user provides an initial prompt to the generator (*step 2* in Figure 3). The initial prompt specifies the essential guidelines to define the CS. They stem from the prescriptive view (Section 3.1), namely the context, the metamodel (or sub-metamodel if slicing is

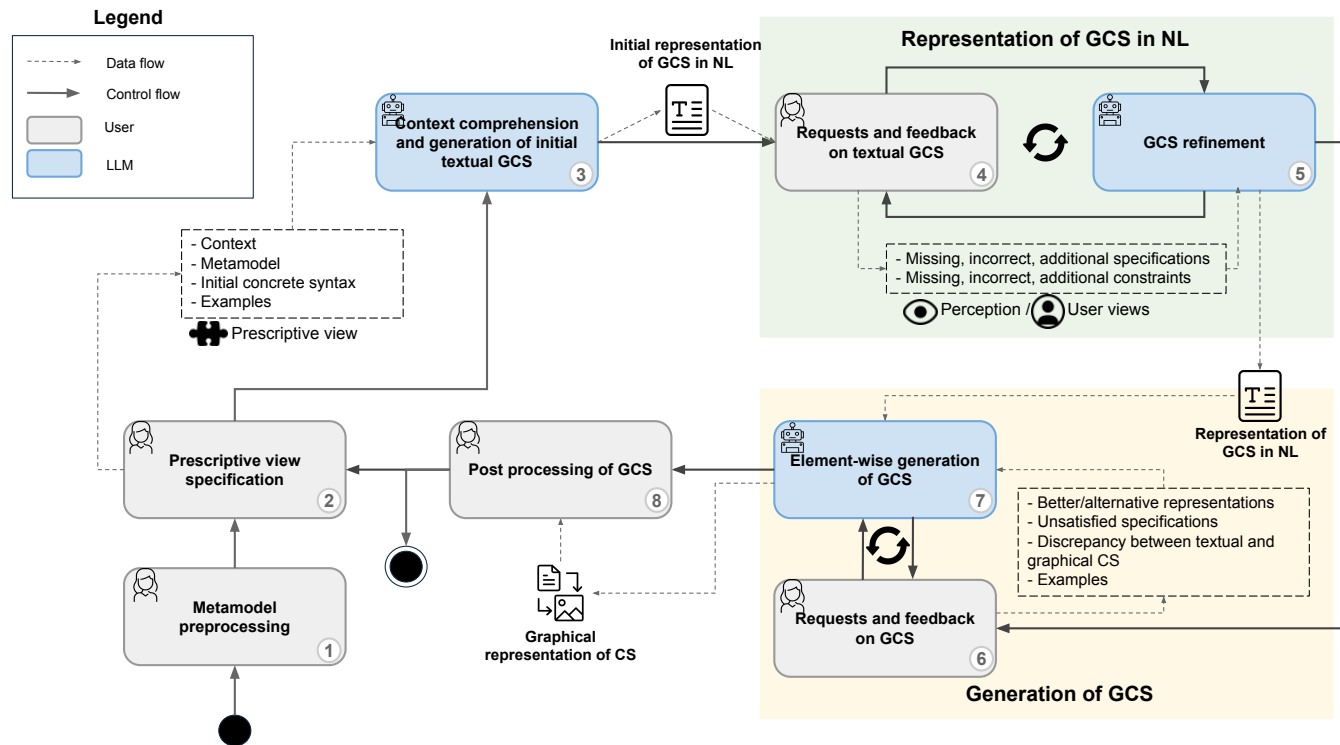


Figure 3: Baseline process for the intelligent generation of tailored concrete syntax using LLMs

```

1 @startuml
2   abstract class Topic {
3     String text
4     Tag tag
5   }
6   class MindMap {
7     String name
8     MainTopic mainTopic
9   }
10  class MainTopic extends Topic {
11    List<SubTopic> subTopics
12  }
13  class SubTopic extends Topic {
14    List<SubTopic> subTopics
15  }
16  class Tag {
17    String name
18  }
19  MindMap "1" *--> "1" MainTopic : Contains
20  MainTopic "1" *--> "*" SubTopic : Contains
21  SubTopic "*" *--> "*" SubTopic : Contains
22  Topic "*" --> "1" Tag : Has
23 @enduml

```

Listing 1: Mind map metamodel in PlantUML syntax

performed), variability, and an initial CS, along with examples, if possible. The more detailed components are described, the more accurate the output is.

For instance, in the mind map example shown in Table 1 *step 2*, the user prompts the LLM by providing the context, specifically,

the scope and the metamodel of the mind map in PlantUML. We explicitly specify that all the metamodel concepts should be included in the generated CS to describe the scope. We also ask the LLM to describe each element of the generated CS in natural language. Recall that we target an intermediate textual phase within our process to help the LLM generate a more accurate output.

4.3 Step 3: Generation of Textual GCS Draft

This initial prompt provides the generator with the task. It allows it to generate a reasonably conceptualized first textual draft of the concepts of the CS (*step 3* in Figure 3). Step 3 of Table 1 reports the generated textual output for the mind map example. This output describes the structure of the mind-map CS by identifying each concept of the CS along with a description of its graphical representation. The description defines each concept’s shape, size, content, and color and its relationships with other concepts.

4.4 Steps 4 and 5: Representation of GCS in NL

Once the generator outputs a first draft, the user can start refining their request to tailor the generated textual representation to their needs and personal preferences by correcting or adding more specifications and constraints (*step 4* in Figure 3). These specifications/constraints are related to the perception and user views of our framework (Sections 3.2 and 3.3). Thus, multiple refinement rounds can be performed until the user is satisfied with the textual description that the LLM provides (*steps 4 and 5* in Figure 3).

Table 1: Excerpt of the conversation with ChatGPT applying the baseline process to the mind map DSML




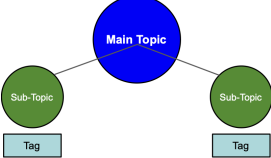




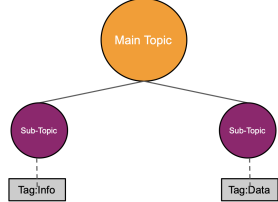


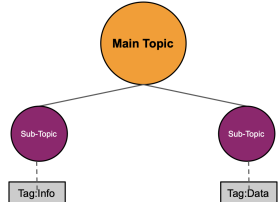

#	Step	Author	Prompt/Output
1	Preprocessing (1)	User	We use PlantUML to textually represent the mind map metamodel.
2	Initial prompt (2)	User	Given the below PlantUML metamodel, generate its concrete syntax by assigning one graphical element for each element of the metamodel. Describe in natural text the generated concrete syntax. <Mind map metamodel in PlantUML>
3	Initial CS (3)	ChatGPT	<ul style="list-style-type: none"> • Main Topic: Represented by a large circle that contains the name of the MindMap at the top [...] • SubTopic: Illustrated as smaller circles or nodes branching out from the MainTopic or other SubTopics, each labeled with its text and associated Tag [...] • Tag: Represented by a colored badge or icon placed next to the text of a Topic, with the Tag name inside. • Has and Contains connections: Illustrated by lines connecting MindMap to MainTopic and MainTopic to SubTopics, with arrowheads indicating the direction of the relationship.
4	Feedback (4)	User	This concrete syntax has few problems as we cannot visually distinguish between the sub-topic and the main topic. How can we adapt it to overcome these problems?
5	Refinement (5)	ChatGPT	<ul style="list-style-type: none"> • Main Topic - Shape: Represent the Main Topic as a larger circle to emphasize its central role. Color: Use a distinct color for the Main Topic circle to set it apart from other elements. Text: Place the text inside the circle, using a bold font to highlight its importance. • Sub-Topic - Shape: Depict Sub-Topics as smaller circles compared to the Main Topic, indicating their supportive role. Color: Choose a different color for Sub-Topics to visually separate them from the Main Topic. Text: The text inside the circle should be in a regular font, distinguishing it from the Main Topic's bold font. Connection: Draw lines from the Main Topic to each Sub-Topic to illustrate the hierarchy. • Tag - Shape: Represent Tags as small rectangles or diamonds to differentiate them from topics. Color: Assign a unique, perhaps lighter or pastel, color palette for Tags [...] Text: Include the tag's text inside the shape, using a smaller font size. Placement: Position the Tag close to its associated topic, perhaps with a connecting line or visually nested under the topic to show the association clearly.
6	Request (6)	User	Given the description of the main topic: <Description> . Generate a graphical representation of this element.
7	Generation (7)	ChatGPT	
8	Request (6)	User	Given the description of the sub-topic: <Description> . Generate a graphical representation of this element.
9	Generation (7)	ChatGPT	
10	Request (6)	User	Given the description of the tag: <Description> . Generate a graphical representation of this element.
11	Generation (7)	ChatGPT	
12	Request (6)	User	Given the description of connections: <Description> . Generate their graphical representation.
13	Generation (7)	ChatGPT	
14	Feedback (6)	User	If we have a large and complex mindmap, tags could be confusing as we might be not able to know to which topic they are associated. How can we modify the above representation to overcome this?

Table 1 (continued): Excerpt of the conversation with ChatGPT applying the baseline process to the mind map DSML

#	Step	Author	Prompt/Output
15	Generation (7)	 ChatGPT	
16	Feedback (6)	 User	Blue and green colors may appear similar for color-blind people. Could you use other colors that are color-blind-friendly?
17	Generation (7)	 ChatGPT	
18	Feedback (6)	 User	The text of the main topic is unreadable. Can you make it more accessible and clear?
19	Generation (7)	 ChatGPT	
20	Postprocessing (8)	 User	We ask ChatGPT to encode the generated graphic in SVG format. It generated the SVG and also a python script that creates it using matplotlib library.

In our mind map example in Table 1, the user points out the difficulty of visually distinguishing between the Main Topic and the Sub-Topic in the initial textual representation. As a response, the generator proposed an improved CS to solve this problem. The improved CS explicitly differentiates between these two elements by assigning different sizes, colors, and fonts to each concept.

4.5 Steps 6 and 7: Generation of GCS

Once the user is satisfied with the textual representation of the CS, they can start asking the LLM to generate the graphical CS element by element (*step 6* in Figure 3). For each generated graphical representation (*step 7* in Figure 3), the user can keep refining their request by 1) asking for alternative representations, 2) treating unsatisfied specifications, 3) pointing out discrepancies between textual and graphical representations, or 4) providing examples.

For the mind-map example in Table 1, steps 6 through 13 show the user asking for a graphical representation for each metamodel concept and the graphical representations generated by ChatGPT. Steps 14 through 19 present multiple refinement rounds that the user suggested to enhance the generated representations.

In this iterative process, the user keeps providing feedback through refinement. Thus, the user is assumed to be satisfied with the final generated syntax. However, when using CS in practice, quantitative feedback (e.g., usability metrics and error rates) could be used to

refine the generated output further. For example, an intelligent generation framework tightly integrated into a modeling environment could suggest updates to particular icons if they are often used erroneously or confused with others.

After the refinement rounds, if the response of the LLM still misses specific nuances, the user can provide examples in the prompt (i.e., few-shot learning [60]). This involves providing the LLM with a small number of carefully selected examples to guide and stimulate the model’s creativity.

4.6 Step 8: Post-processing

Once the user is satisfied with the generated output, the result should be post-processed to be integrated into a modeling environment. For instance, in our baseline process, the user saves the icons individually in the desired format and uploads them in their modeling tool. This step can be automated by integrating the API calls in the modeling tool.

For the mind map example in Table 1, the user asks the LLM to write a Python script to export the final icons in *SVG* format. The user can then upload these icons in their modeling tool.

5 DISCUSSION

This section discusses applying the baseline process to the mind-map example to illustrate the approach's benefits. We then investigate limitations and research challenges discovered when applying this process to a more complex DSML of traffic lights.

5.1 Success for Simple DSMLs

As shown in Section 4, applying the baseline process using LLMs to a simple domain-specific language, i.e., mind-map, works reasonably well. This suggests that our approach can generate CS for straightforward small-sized applications that incorporate basic user preferences and accommodate specific needs like customizing various aspects such as font size, node shapes, and line styles.

We discovered that the key to a successful generation is ensuring the context is well understood to ensure that the generated CS is semantically rich and aligned with the specific challenges and objectives of its intended use case. Our preliminary exploration found that a detailed and precise initial prompt and iterative refinement steps based on user feedback form the cornerstone of effectively leveraging LLMs for DSML development. This process can allow for the customization of syntax to meet functional requirements and enhance the user experience through considerations of accessibility and personalization. However, several feedback iterations were needed to achieve the desired CS.

5.2 Shortcomings for Complex DSMLs

Despite the promising initial results, our exploration into the intelligent generation of tailored CS also highlighted several gaps and challenges. We encountered various limitations and challenges, which were particularly evident when applying our framework to the example of traffic light for individuals with color blindness. In this example, we aimed to model the structure and behavior of traffic signals at an intersection, involving the lights, timing for changes, and modes for emergency vehicles. This DSML requires adaptations to ensure that individuals with color vision deficiencies can accurately interpret and respond to traffic signals, thus enhancing both safety and accessibility. We chose this example to test our baseline process on a more sophisticated scenario, intending to identify key areas for future focus in generating tailored CS.

5.2.1 Process application to traffic light Following the iterative approach outlined earlier, we first focus on the comprehension of the context of our system by LLMs, specifically ChatGPT4, by providing the metamodel in PlantUML. This metamodel defines concepts like the state of the lights, modes, etc. Additionally, we articulate the need to accommodate individuals with color vision deficiencies in natural language, setting the stage for a tailored CS generation. Once we achieve the textual generation and obtain the desired description for the traffic light CS along with details about each element, we move to graphical generation. We explore two distinct approaches: generating the entire CS at once and generating the CS for language segments.

5.2.2 Attempt 1: Entire concrete syntax generation Initially, we tasked the generator with producing the graphical CS in a single comprehensive attempt, which produced graphical symbols for

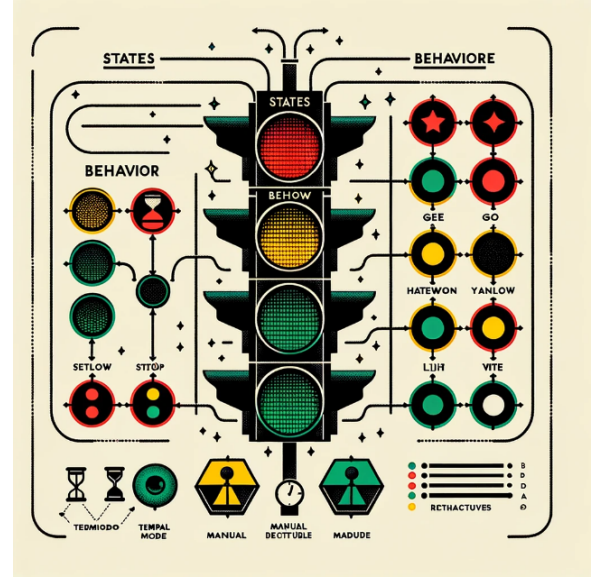
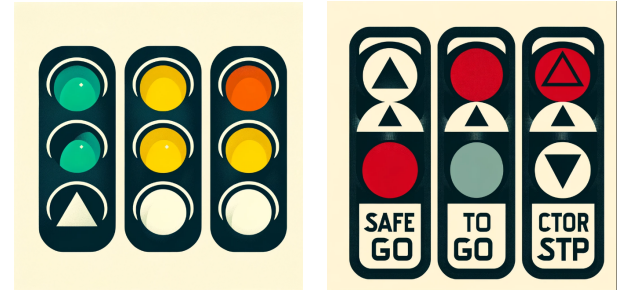


Figure 4: Attempt 1: Result of producing the entire graphical concrete syntax in one step.



(a) Step i- Results of the prompt: "Given the description of light elements, generate a graphical representation for these elements."

(b) Step i+1- Results of the prompt: "Given the description of behavior elements, generate a graphical representation for these elements."

Figure 5: Attempt 2: Different outputs of iterative steps of traffic light system's tailored concrete syntax.

every element. Through this attempt, we wanted to evaluate the generator's capacity to integrate multiple aspects of our framework—such as color adaptations for color vision deficiencies and symbolic representation—into a cohesive visual output.

The final output of this attempt is depicted in Figure 4. The obtained CS offers some insightful directions, but it certainly does not meet our expectations. The image generation process cannot focus on each graphical element in turn. Instead, it generates an unusable mix of unlabelled elements that cannot be divided up into individual icons.

5.2.3 Attempt 2: Segmented concrete syntax generation In the second attempt, we segmented the process similar to the mind map example in Section 4. We instructed the generator to focus on each

conceptual element separately, ensuring each was clearly defined and accessible. Figure 5a presents the results where we specifically request graphical representations focusing solely on the light elements. The obtained results are confusing and not explanatory. Figure 5b presents the results where we ask ChatGPT4 to generate from the description of the behavioral traffic light elements. The output fell short of our expectations, although we focused on a few metamodel elements.

5.2.4 Concerns with iterative feedback loops A key concern with the feedback loop in the iterative process of metamodel refinement is determining when to terminate interactions with the human user. The iterations conclude when the metamodel is fully covered, and the specifications are satisfied, but the number of iterations required is inherently uncertain. It is a fuzzy problem that depends on various factors, including context, scope, variability, problem complexity, the results generated by the LLM, and the user's subjective preferences. However, we expect that the LLM-human interactions will progressively improve the output. By incorporating additional context information (e.g., feedback, domain knowledge, and symbolic reasoning), the LLM can be guided toward improved solutions. As generative AI techniques continue to evolve, we expect a reduction in the number of iterations needed, ideally achieving the ultimate goal of generating graphical CS in a single iteration based on the entire specification. Based on our experiments, the number of iterations varied significantly. The mind-map case required three to five iterations (request/generation) due to its simpler structure, semantics, and fewer elements. For a complex example (i.e., traffic light), seven to ten iterations were necessary due to its greater complexity, involving more intricate behaviors, detailed interactions, and rich semantics. Thus, more complex scenarios require more iterations to achieve satisfactory results.

5.3 Research Challenges

The inconsistency of the obtained results across the two explored case studies, i.e., mind map and traffic light languages, highlights that the effectiveness of our suggested process can vary significantly with the DSML's complexity. This motivates the following research challenges in generating tailored CS for more intricate systems.

Specifically, we identify some research challenges we foresee in creating and implementing an intelligent CS generation framework. These challenges were a) identified and collected by the authors while performing the baseline process on these two DSML examples and b) discussed and organized by all authors.

RC1: Capturing and providing language details A primary challenge is capturing the full semantics of the DSML in the generated CS, especially for complex domains. As we highlighted earlier, the process relies heavily on the comprehensiveness of the initial prompt and the effectiveness of the feedback loop in conveying the nuanced requirements of the DSML to the LLM. This remains a challenge for a very complex system.

The reliance of the process on LLMs also introduces the challenge of managing the complexity of prompts. There is an assumption that providing more detailed and comprehensive information results in better outcomes [33]. However, overly complex prompts

can overwhelm the LLM, leading to less effective or irrelevant outputs [26, 54]. This balance between detail and clarity in prompts is a critical factor in the success of the generation process.

RC2: Automating and standardizing feedback For the intelligent generation of syntax, the user's presence is crucial to provide feedback and validate the compliance of the generated results. In the proposed process, the feedback process is highly manual. Currently, the user is solely responsible for assessing the quality of the AI-generated results with respect to the diverse attributes encompassed within our framework. This user-centric approach is subjective and significantly burdens individuals, necessitating their constant engagement to ensure the results align with the different attributes of our framework. Therefore, there exists a need for automated methods that can capture user preferences and validate the generated results automatically. For instance, leveraging other LLMs as validators and providers of feedback presents a viable strategy [32, 53]. LLMs could potentially automate the feedback and validation processes by simulating user behavior.

RC3: Intelligent agent image understanding and generation Recent research in natural language processing has witnessed unprecedented progress. Integrating LLMs with computer vision models enabled the development of powerful and sophisticated chatbots (e.g., ChatGPT, Claude). These chatbots demonstrate remarkable proficiency in textual comprehension and generation. However, their image understanding and generation capabilities, particularly for complex and abstract images, remain relatively limited [49]. Further advancements in this field would enable the direct generation of more advanced concrete syntaxes for complex metamodels. This may also allow for direct refinement and feedback steps on graphical representations without the necessity for intermediate textual generation.

RC4: Segmentation and integration of the metamodel In the proposed process, we advised breaking down the problem into smaller sub-problems and including only a fragment of the metamodel that requires changes, particularly in the case of complex metamodels. This approach would simplify the problem for LLMs, streamline prompts, and improve outcomes. However, several considerations should be taken into account, such as the potential risk of losing the context, consistency, or integrity of the overall metamodel when focusing on isolated fragments. Moreover, the granularity at which the metamodel and problem are divided is important, as it helps to balance oversimplification and maintaining a manageable level of complexity. Therefore, there is a need for a systematic approach to segment this metamodel and to divide the problem into smaller sub-problems [8, 26].

RC5: Tooling integration and automation Users need intuitive and robust tooling to use our envisaged intelligent CS generator. We see multiple research questions on integrating this intelligent generation within modeling tools. For example, qualitative and quantitative user studies will be required to provide a streamlined UX for user interaction with the LLM. An example is providing prompt templates or low-code approaches for the user to easily and reliably prompt the LLM, such as ChainForge [3]. There are also intersections with privacy concerns, as the user's disabilities should not be revealed to third-party services.

RC6: Evaluation metrics to consider Another significant challenge is the absence of a standardized benchmark or validation mechanism to evaluate the completeness and efficacy of the generated syntax. This makes it challenging to objectively assess when the generated syntax meets the intended design goals, potentially leading to an extended iterative process without clear direction. Consequently, there is a need to design quantitative metrics to evaluate the attributes of our framework. This would facilitate the objective assessment of our framework (e.g., develop metrics that measure the accessibility of the CS and the visual expressiveness of its attributes). The process would become more efficient and less dependent on subjective user evaluations.

To address these challenges, we propose the following metrics for evaluating the automated generation of graphical CS:

- **Computation Time and Resources:** Evaluate the efficiency of the generation process by measuring the time taken, computational resources needed by the LLM, number of prompts/tokens required, number of iterations, etc.
- **Coverage:** Ensure all metamodel elements are represented in the graphical CS. The approach tooling can automatically check this based on the post-processing step.
- **User Satisfaction:** Collect qualitative feedback from domain experts and end users to measure how well the generated syntax meets their needs and expectations.
- **Experimental Results:** Compare the resulting CS against existing CS by having users perform a series of experiments with both. Collect qualitative and quantitative metrics to determine which syntax users prefer and which leads to faster and more accurate experiment solutions.
- **Artificial Evaluation:** Evaluate the CS using a multi-modal language model. For instance, a query could determine if the icons are culturally appropriate. This fits within a generator/adversary paradigm to automatically evolve concrete syntaxes, as suggested by a reviewer.

6 CONCLUSION

In this paper, we presented a vision for CS generation in MDE. First, we proposed a multi-view conceptual framework for generating CS tailored for specific usage. This framework integrates the prescriptive view of language (e.g., context and meta-model), the perception view (i.e., graphical language and user interface principles), the user view (i.e., preferences and considerations), and user-provided examples. Second, we introduced a baseline process to use our framework with LLMs in practice. We instantiate this process using the example language of the mind-map to showcase the application of our framework. Finally, we discussed the traffic lights example to highlight the limitations and research challenges related to the intelligent generation of CS using current LLM technology.

Our findings show the potential of state-of-the-art LLMs to generate tailored CS. However, several limitations are immediately apparent when applying our framework to a more intricate problem. It may be possible to use our approach to generate the full CS from simple cases, but it can be used to explore different CS ideas and concepts for more complex cases.

Our future work is to address some of the research challenges identified in Section 5.3. In particular, we aim to develop an open-source intelligent CS generator as a platform for future research and development. As a first step, this platform should offer state-of-the-art modeling guidance [10] to aid the user along the baseline process presented in Section 4 to generate the graphical CS. We will then conduct systematic user studies to improve the UX of the platform and the generated CS. Further, we plan to extend our platform to accommodate textual CS generation.

ACKNOWLEDGMENT

This article originated from the *Software Engineering at Montréal (SEMTEL)* research meetings, which regularly bring together the software engineering research community in Montréal, Canada. Find out more at <https://semte.github.io/>.

REFERENCES

- [1] L. Almonte et al. 2021. Automating the synthesis of recommender systems for modelling languages. In *Software Language Engineering*. 22–35.
- [2] D. Amyot, H. Farah, and J. Roy. 2006. Evaluation of development tools for domain-specific modeling languages. In *System Analysis and Modeling*.
- [3] I. Arawjo et al. 2023. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. *arXiv preprint arXiv:2309.09128* (2023).
- [4] S. Arulmohan, M. Meurs, and S. Mosser. 2023. Extracting domain models from textual requirements in the era of large language models. In *Model Driven Engineering Languages and Systems Companion*. IEEE, 580–587.
- [5] T. Baar. 2008. Correctly defined concrete syntax. *Software & Systems Modeling* 7 (2008), 383–398.
- [6] B. Barricelli et al. 2019. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software* 149 (2019), 101–137.
- [7] A. Blouin et al. 2014. Slicing-Based Techniques for Visualizing Large Metamodels. In *Working Conference on Software Visualization*. 25–29.
- [8] A. Blouin et al. 2015. Assessing the use of slicing-based visualizing techniques on the understanding of large metamodels. *Information and Software Technology* 62 (2015), 124–142.
- [9] M. B. Chaaben, L. Burgueño, and H. Sahraoui. 2023. Towards using few-shot prompt learning for automating model completion. In *International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE, 7–12.
- [10] S. Chakraborty and G. Liebel. 2024. Modelling guidance in software engineering: a systematic literature review. *Software & Systems Modeling* 23, 1 (2024), 249–265.
- [11] K. Chen et al. 2023. Automated Domain Modeling with Large Language Models: A Comparative Study. In *Model Driven Engineering Languages and Systems*. IEEE, 162–172.
- [12] H. Cho, J. Gray, and E. Syriani. 2012. SyntaxMap: A Modeling Language for Capturing Requirements of Graphical DSL. In *Asia-Pacific Software Engineering Conference*, Vol. 1. IEEE, 705–708.
- [13] S. Costa et al. 2021. Ontologies in human-computer interaction: A systematic literature review. *Applied Ontology* 16, 4 (2021), 421–452.
- [14] D. Deeptimahanti and R. Sanyal. 2011. Semi-automatic generation of UML models from natural language requirements. In *India Software Engineering Conference*. 165–174.
- [15] K. D. Dhole, R. Chandradevan, and E. Agichtein. 2023. An interactive query generation assistant using LLM-based prompt modification and user feedback. *arXiv preprint arXiv:2311.11226* (2023).
- [16] S. Ekin. 2023. Prompt engineering for ChatGPT: A quick guide to techniques, tips, and best practices. *Authorea Preprints* (2023).
- [17] Frédéric Fondement. 2007. *Concrete syntax definition for modeling languages*. Ph.D. thesis. Ecole Polytechnique de Lausanne.
- [18] F. Fondement and T. Baar. 2005. Making Metamodels Aware of Concrete Syntax. In *Model Driven Architecture – Foundations and Applications*. Springer, 190–204.
- [19] W. O. Galitz. 2007. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. Number 3rd Edition. Wiley.
- [20] T. R. G. Green. 1989. Cognitive Dimensions of Notations. In *People and Computers V*. British Computer Society Workshop Series, 443–460.
- [21] Fill Hans-Georg, Peter Fette, and Julius Köpke. 2023. Conceptual Modeling and Large Language Models: Impressions From First Experiments With ChatGPT. *Enterprise Modelling and Information Systems Architectures* 18, 3 (2023), 1–15.
- [22] D. Harel and B. Rumpe. 2004. Meaningful Modeling: What’s the Semantics of “Semantics”? *IEEE Computer* 37, 10 (2004), 64–72. <https://doi.org/10.1109/MC.2004.172>

- [23] Z. Hu et al. 2023. Unlocking the Potential of User Feedback: Leveraging Large Language Model as User Simulator to Enhance Dialogue System. *arXiv preprint arXiv:2306.09821* (2023).
- [24] S. Karim and A. Tjoa. 2006. Towards the use of ontologies for improving user interaction for people with special needs. In *International Conference on Computers for Handicapped Persons*. Springer, 77–84.
- [25] S. Kelly, K. Lyytinen, and M. Rossi. 1996. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *Advanced Information Systems Engineering: 8th International Conference, CAISE'96 Heraklion, Crete, Greece, May 20–24, 1996 Proceedings 8*. Springer, 1–21.
- [26] T. Khot et al. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406* (2022).
- [27] D. S. Kolovos et al. 2010. Taming EMF and GMF using model transformation. In *Model Driven Engineering Languages and Systems*. Springer, 211–225.
- [28] T. Kosar, S. Bohra, and M. Mernik. 2016. Domain-specific languages: A systematic mapping study. *Information and Software Technology* 71 (2016), 77–91.
- [29] S. Kuhar and G. Polancic. 2021. Conceptualization, measurement, and application of semantic transparency in visual notations. *Software & Systems Modeling* 20 (2021), 2155–2197.
- [30] T. Kühne. 2006. Matters of (Meta-)Modeling. *Software & Systems Modeling* 5, 4 (2006), 369–385.
- [31] S. K. Lahiri et al. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950* (2022).
- [32] H. Lee et al. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267* (2023).
- [33] P. Lewis et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
- [34] B. Li et al. 2023. Eliciting human preferences with language models. *arXiv preprint arXiv:2310.11589* (2023).
- [35] L. Luque et al. 2014. Can we work together? on the inclusion of blind people in uml model-based tasks. In *Inclusive Designing: Joining Usability, Accessibility, and Inclusion*. Springer, 223–233.
- [36] G. Marcus, E. Davis, and S. Aaronson. 2022. A very preliminary analysis of DALL-E 2. *arXiv preprint arXiv:2204.13807* (2022).
- [37] B. Mariño et al. 2018. Accessibility and activity-centered design for ICT users: ACCESSIBILITIC ontology. *IEEE Access* 6 (2018), 60655–60665.
- [38] O. Metatla, N. Bryan-Kinns, and T. Stockman. 2008. Constructing relational diagrams in audio: the multiple perspective hierarchical approach. In *Computers and accessibility*. 97–104.
- [39] D. Moody. 2009. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (2009), 756–779.
- [40] F. Mu et al. 2023. ClarifyGPT: Empowering LLM-based Code Generation with Intention Clarification. *arXiv preprint arXiv:2310.10996* (2023).
- [41] P. Muller, P. Studer, and J. Jézéquel. 2004. Model-driven generative approach for concrete syntax composition. In *OOPSLA Workshop on Best Practices for Model-Driven Development*.
- [42] B. A. Nardi and C. L. Zamer. 1993. Beyond models and metaphors: Visual formalisms in user interface design. *Journal of Visual Languages & Computing* 4, 1 (1993), 5–33.
- [43] B. Nastov and F. Pfister. 2014. Experimentation of a Graphical Concrete Syntax Generator for Domain Specific Modeling Languages.. In *INFORSID*. 197–213.
- [44] A. Pescador and J. de Lara. 2016. DSL-maps: from requirements to design of domain-specific languages. In *Automated Software Engineering*. 438–443.
- [45] A. Rahimi et al. 2023. Towards Generating Structurally Realistic Models by Generative Adversarial Networks. In *Model Driven Engineering Languages and Systems Companion*. IEEE, 597–604.
- [46] Z. Rajaei et al. 2021. A DSL for Encoding Models for Graph-Learning Processes. In *Workshop on OCL and Textual Modeling*.
- [47] A. Ramesh et al. 2021. Zero-shot text-to-image generation. In *International conference on machine learning*. PMLR, 8821–8831.
- [48] I. Ráth, A. Ökrös, and D. Varró. 2010. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software & Systems Modeling* 9 (2010), 453–471.
- [49] K. I Roumeliotis and N. D. Tselikas. 2023. ChatGPT and Open-AI models: A preliminary review. *Future Internet* 15, 6 (2023), 192.
- [50] A. Sadik, S. Brulin, and M. Olhofer. 2023. *Coding by Design: GPT-4 empowers Agile Model Driven Development*. Report 2310.04304. arXiv.
- [51] A. Sarioğlu, H. Metin, and D. Bork. 2023. How inclusive is conceptual modeling? A systematic review of literature and tools for disability-aware conceptual modeling. In *International Conference on Conceptual Modeling*. Springer, 65–83.
- [52] T. Schick and H. Schütze. 2021. Few-Shot Text Generation with Natural Language Instructions. In *Empirical Methods in Natural Language Processing*. ACL, 390–402.
- [53] O. B. Sghaier and H. Sahraoui. 2024. *Improving the Learning of Code Review Successive Tasks with Cross-Task Knowledge Distillation*. Report 2402.02063. arXiv.
- [54] S. Suh et al. 2023. Sensecape: Enabling multilevel exploration and sensemaking with large language models. In *User Interface Software and Technology*. 1–18.
- [55] E. Syriani et al. 2013. AToMPP: A web-based modeling environment. In *Model Driven Engineering Languages and Systems*, Vol. 1115. CEUR-WS.org, 21–25.
- [56] C. Tinnes et al. 2023. *Towards Automatic Support of Software Model Evolution with Large Language Models*. Report 2312.12404. arXiv.
- [57] J. Van Gorp, J. Bosch, and M. Svahnberg. 2001. On the notion of variability in software product lines. In *Working Conference on Software Architecture*. 45–54.
- [58] Y. Van Tendeloo et al. 2017. Concrete Syntax: A Multi-Paradigm Modelling Approach. In *Software Language Engineering*. ACM, 182–193.
- [59] V. Viović, M. Maksimović, and B. Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *Intelligent Engineering Systems*. IEEE, 233–238.
- [60] Y. Wang et al. 2020. Generalizing from a few examples: A survey on few-shot learning. *Comput. Surveys* 53, 3 (2020), 1–34.
- [61] J. Wei et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
- [62] M. Weyssow, H. Sahraoui, and E. Syriani. 2022. Recommending metamodel concepts during modeling activities with pre-trained language models. *Software & Systems Modeling* 21, 3 (2022), 1071–1089.
- [63] M. Wolfgang. 2006. *Laws of Seeing*. MIT Press.
- [64] L. Wouters. 2013. Towards the notation-driven development of DSMLs. In *Model-Driven Engineering Languages and Systems*, Vol. 16. Springer, 522–537.
- [65] S. Yang and H. Sahraoui. 2022. Towards automatically extracting UML class diagrams from natural language specifications. In *Model Driven Engineering Languages and Systems Companion Proceedings*. 396–403.
- [66] S. Zhang et al. 2023. *Instruction Tuning for Large Language Models: A Survey*. Report 2308.10792. arXiv.