Automatically Generating Visual Modeling Environments

HUGO DA GIÃO, Faculty of Engineering, University of Porto, Portugal and HASLab / INESC TEC, Portugal
RUI PEREIRA, OutSystems, Portugal
MIGUEL VILAÇA, Universidade Católica Portuguesa, Portugal
JÁCOME CUNHA, Faculty of Engineering, University of Porto, Portugal and HASLab / INESC TEC, Portugal

Model-Driven Engineering (MDE) offers significant benefits but faces usability challenges, particularly in developing intuitive modeling interfaces. Most existing tools rely on textual or tree-based editors, which are often difficult for non-technical users to navigate. Creating custom graphical editors based on existing metamodels, a common way to describe modeling environments and languages, typically requires substantial manual effort and specialized knowledge of complex frameworks such as Sirius. These hurdles limit the broader adoption of MDE among developers and domain experts. Consequently, there is an increasing demand for automated, flexible, and user-friendly approaches to generating interfaces that can make MDE more accessible and practical across various domains.

In this work, we present a method for automatically generating a visual modeling environment from existing metamodels. Based on this method, we devised *Model2Block*, a command-line application that automatically generates block-based visual languages from metamodels defined using the Ecore language. The tool analyzes an input metamodel and produces a corresponding set of blocks, which are then integrated into a web-based environment. This enables users to visually build models using blocks that represent valid model elements.

We present an applicability study demonstrating that our method and tool were successfully applied to a publicly available dataset containing more than 50 metamodels. Moreover, we present a case study of a concrete modeling environment we generated from a metamodel, achieving, in this particular case, higher usability and lower workload than an existing tool.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **Domain specific languages**; **Visual languages**; **Software creation and management**.

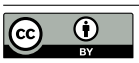
Additional Key Words and Phrases: Model-Driven Engineering, Modeling Languages, Visual Languages, Block-Based Languages, Blockly, Modeling Environments, Graphical User Interfaces, GUI Generation

ACM Reference Format:
Hugo da Gião, Rui Pereira, Miguel Vilaça, and Jácome Cunha. 2026. Automatically Generating Visual Modeling Environments. *Proc. ACM Hum.-Comput. Interact.* 10, 4, Article EICS025 (June 2026), 19 pages. https://doi.org/10.1145/3816777

1 Introduction

Model-Driven Engineering (MDE) is a software development paradigm in which high-level models constitute the central artifacts of the development process. By abstracting system structure and behavior into models, MDE seeks to improve productivity, maintainability, and portability across

Authors' Contact Information: Hugo da Gião, Faculty of Engineering, University of Porto, Porto, Portugal and HASLab / INESC TEC, Porto, Portugal, hugo.a.giao@inesctec.pt; Rui Pereira, OutSystems, Braga, Portugal, rui.alexandre.pereira@outsystems.com; Miguel Vilaça, Universidade Católica Portuguesa, Braga, Portugal, jvilaca@ucp.pt; Jácome Cunha, Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal and HASLab / INESC TEC, Porto, Portugal, jacome@fe.up.pt.



ACM 2573-0142/2026/6-ARTEICS025
https://doi.org/10.1145/3816777

platforms. Central to this paradigm are metamodels, which formally define the syntax and semantics of domain-specific modeling languages using frameworks such as Ecore [46]. Tooling support, notably the Eclipse Modeling Framework (EMF) [46] and model-to-text transformation tools such as Acceleo [42], enables automation across the development lifecycle and underpins many industrial MDE practices [13].

From a Human–Computer Interaction (HCI) perspective, however, the effectiveness of MDE is tightly coupled to the usability of its modeling environments. Despite their expressive power, most MDE tools expose models through textual syntaxes or tree-based editors that require users to navigate abstract structures and strict constraints. These interaction styles are often unintuitive, cognitively demanding, and error-prone, particularly for domain experts without a (strong) software engineering background [6]. As a result, a significant gap emerges between the theoretical benefits of MDE and its practical adoption, rooted not in modeling capabilities but in limitations of interaction design.

Designing custom graphical editors that better align with users' mental models can mitigate these issues, but doing so typically requires substantial manual effort and specialized technical expertise. Frameworks such as Sirius [23] offer powerful capabilities for defining visual syntaxes, yet they introduce additional complexity in terms of configuration, maintenance, and learning curve. From an HCI standpoint, this creates a tension between flexibility and usability: while MDE promotes abstraction and automation, the tools for interacting with models often remain rigid and developer-centric. Consequently, there is a strong need for automated and model-driven approaches to user interface (UI) generation that place interaction design concerns—such as learnability, error prevention, and cognitive load—at the forefront.

Within this context, block-based visual languages [7] provide a compelling interaction paradigm for MDE environments. Blocks represent modeling constructs as tangible visual elements that can be composed through direct manipulation. Their constrained, snap-together behavior implicitly enforces metamodel rules, reducing syntactic errors and providing immediate feedback to users. Such environments have demonstrated strong usability benefits in educational settings, such as Scratch and Code.org [43], as well as in domain-specific configuration tasks, such as workflow modeling and IoT systems [49]. Google's Blockly framework [25] illustrates how language constructs can be defined declaratively and mapped to interactive visual components, in a way conceptually similar to defining a metamodel in MDE.

From an HCI-informed MDE perspective, block-based modeling environments exemplify how visual representations can bridge the gap between formal models and user cognition. By embedding domain constraints directly into the interaction mechanisms, they shift complexity from the user to the tool, aligning with the principle of designing notations optimized for communication and problem solving [41]. Integrating such interaction paradigms into model-driven engineering workflows opens promising directions for making MDE more accessible, especially for domain experts, while preserving the rigor and automation benefits that characterize the paradigm.

With this work, we intend to answer the following research questions (RQs):

RQ1: How can we automatically generate block-based visual modeling environments from metamodels?

Our objective is to define a reusable and automated approach that captures the structure and constraints of a modeling language, expressed as a metamodel, within a generated modeling environment with a concrete syntax that uses blocks as constructs.

Building on Blockly, we propose an approach to automatically generate graphical user interfaces from existing metamodels. Our work leverages block-based programming to produce visual

modeling environments from metamodel specifications. By exploiting Blockly's flexibility, we automatically generate visual blocks representing the elements of the target metamodel and assemble them into a complete web-based modeling interface. This interface enables users to construct models that conform to the metamodel without requiring textual syntax or prior knowledge of modeling frameworks, improving both accessibility and usability.

Our method supports the generation of robust, user-friendly interfaces without requiring modifications to the original metamodels, thereby preserving the integrity of existing domain models. An example of a generated interface is shown in Figure 3.

We describe this process in Section 4 and present a prototype implementation in Section 5. Our hypothesis is that an automated algorithmic approach can be designed and implemented to parse and map all concepts represented in Ecore metamodels into a compatible Blockly representation.

RQ2: Can the method we devised be used to generate correct visual modeling environments from arbitrary metamodels?

We evaluate the applicability of our approach and the correctness of the generated environment by applying our prototype tool to an independent dataset containing 53 metamodels [9], examining the completeness of the generated blocks, and testing their integration in a web-based environment. This evaluation is discussed in Section 6.

This study shows that our proposal can consistently produce a modeling environment for a given metamodel without manual intervention.

RQ3: Can the generated visual modeling environments improve usability?

Finally, to evaluate the usability of the generated environments, we conducted a user study in which participants compared the usability and workload of a generated environment for a particular domain with those of existing tools. A summary of this study is presented in Section 7.

As shown, the participants rated the environment generated by our approach (using the System Usability Scale (SUS) [14]) as more user-friendly than the most commonly used tool. They also felt the workload of the generated interface was lower.

Based on the answers to our RQs, our paper provides the following contributions:

(1) A method for automatically generating visual modeling environments for any given modeling language.
(2) A mapping between the Ecore language and a block-based environment.
(3) A prototype tool termed *Model2Block* that can automatically produce a visual modeling environment based on a given metamodel.

In the remainder of this paper, we present related work in Section 3 and our conclusions and future work in Section 9.

2 Motivational Example

In this section, we provide a running example to showcase our approach.

Consider the metamodel illustrated in Figure 1, which represents a language for constructing continuous integration and delivery (CI/CD) pipelines [28], that is, scripts that aid software engineers in developing and delivering software faster and reliably [33]. Simplistically, and ignoring part of the model/language, a pipeline is formed by a set of jobs that, in turn, execute a set of commands within a tool (e.g., an operating system). These concepts are represented by the classes surrounded in red dashed lines.

To create CI/CD pipelines using this language, one needs to build a model in a modeling environment. This can be done using a concrete syntax (textual or graphical) specifically designed for the purpose. However, to define such a syntax is a manual, time-consuming task. Another possibility is

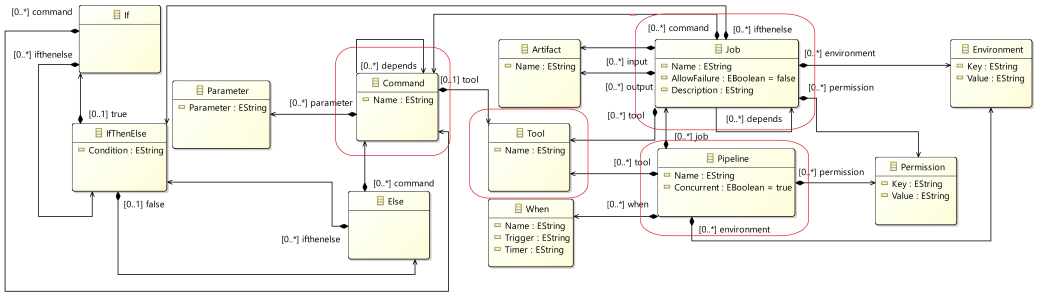


Fig. 1. Example of a metamodel defining a language for creating CI/CD pipelines.

to use a general-purpose modeling framework (e.g., EMF), which provides tree- or tabular-based model visualizations. However, these general environments have a technical look-and-feel, may not be the most suitable for domain experts, and often require the use of a heavy-duty modeling environment.

On the other hand, block-based languages [7] have been used quite successfully, particularly in environments where users often lack a computer science background, as shown in Section 3. Thus, if we consider that each concept in the language, i.e., each class in the metamodel, can be represented by a block, and that the relations between these classes can be encoded by how these blocks fit together, we can define a visual language to create models of this language.

Suppose we use the general block shown in Figure 2 to represent a class C with attributes $a1$ and $a2$. We would then create a concrete block for the class Pipeline, another for Job, and so on. Figure 3.A shows the Pipeline and Job blocks.

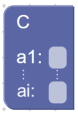


Fig. 2. A generic block to construct a block-based concrete syntax for a metamodel-defined language.

Block-based languages also allow us to define how blocks can be put together. Since we know a pipeline is composed of jobs, we can define this constraint in the block-based language. Figure 3.B shows a concrete pipeline being created. It consists of a Pipeline named *default_pipeline*, which in turn consists of a Job named *build*. In fact, these blocks cannot be used in any other way. This ensures the models created follow the rules established by the metamodel.

This environment was, in fact, automatically generated by our approach and the prototype we implemented. Moreover, it includes other features, as we shall discuss in Section 5.3.

3 Related Work

3.1 Modeling Environments and Frameworks

Model-driven engineering has a long tradition of employing graphical environments to bring domain experts closer to formal modeling, as evidenced by comprehensive surveys of MDE tool support [45]. Early environments such as the Eclipse Graphical Modeling Framework (GMF) [22] and Eclipse Sirius [23] provided rich drag-and-drop editors tailored to Ecore metamodels, but required significant manual effort in palette and tooling configuration [24]. On the other hand, we

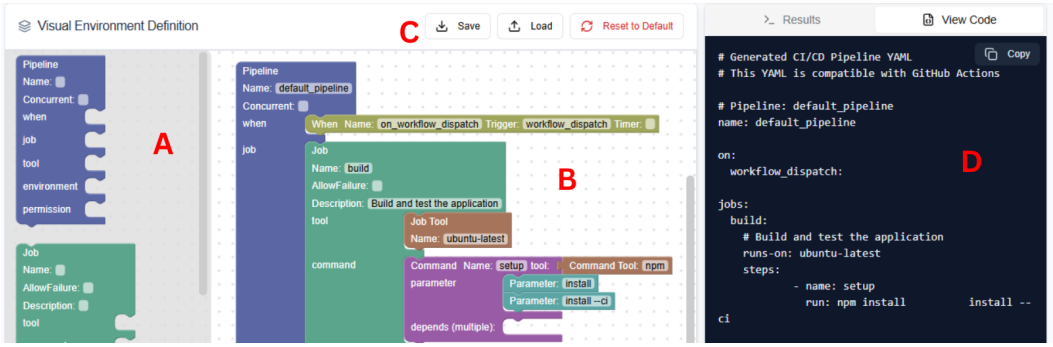


Fig. 3. Web-based visual environment showcasing the blocks generated from CI/CD metamodel.

propose a fully automated tool for generating visual languages that have proven successful among their users.

Research on variability injection demonstrates how EMF/Sirius environments can be automatically adapted to support visual product line engineering [26]. By contrast, our focus lies on block-based visualization approaches rather than extending existing Sirius-based editors.

Blended metamodeling has demonstrated that combining graphical and natural-language inputs can foster collaboration between technical and non-technical stakeholders, easing communication and co-design [3]. However, these frameworks primarily focus on the definition of the languages, that is, the metamodels, rather than on the concrete syntax for domain experts, which is the goal of our work.

Several cloud-based platforms try to minimize the effort of modelers. MDENet simplifies this learning curve for MDE newcomers by offering an online playground for exploring MDE concepts without requiring tool installation [51]. Jjodel is a reflective platform addressing cognitive complexity and usability challenges commonly encountered in MDE tools. It empowers language designers to define and refine domain-specific languages with minimal overhead. By leveraging modern front-end technologies, Jjodel bridges theoretical MDE research and practical applications, providing an accessible solution for diverse modeling contexts [15]. Dandelion is another example of a cloud-native, scalable graphical modeling platform built on MDE principles. It enables harmonized metamodel representations and lazy model loading to support industrial-scale low-code development [39]. BESSER is yet another online framework for defining models. It is based on the Unified Modeling Language (UML) [16] and supports multiple model types, ranging from structural (similar to UML class diagrams) to state machines [16]. However, these frameworks focus on the modelers and their needs, which makes them quite interesting for defining and handling models and metamodels. Unfortunately, they provide little to no support in terms of concrete syntax for the metamodels/languages defined using them, which is precisely our goal.

3.2 Domain-Specific Visualization and Interaction

Additional approaches emphasize domain-specific visualization and natural interaction modalities. Bottoni et al. pioneered graph-grammar–based methods for specifying editing and execution patterns tied to metamodel rules, enabling formal interaction control [11, 12]. Draheim et al. introduced visual reification, providing intuitive domain-specific visualizations for non-technical users [20, 21]. However, these approaches still require some manual definition of the concrete syntax, whilst ours is fully automatic.

Lumertz et al. developed a graph-based metamodel for modular user interface design [37], proposing a structured approach in which UI components are represented as graph elements to improve reusability, composability, and consistency across applications. Ziegler et al. extended visualization to simulation-driven metamodels [50], enabling dynamic visual feedback that supports analysis and validation of complex system behavior during simulation. López-Jaquero et al. automated gesture-driven editor generation [38], introducing model-based techniques that transform high-level specifications into interactive, gesture-enabled editing environments. Domokos and Varró contributed an SVG-based rendering framework [19], facilitating scalable, platform-independent visualization of models through vector-based graphical representations. Gebhardt et al. presented memoSlice, an interactive multi-view visualization tool for high-dimensional metamodels [27], which supports coordinated views, filtering, and slicing to help users explore complex model structures more effectively. Similarly, Blouin et al.'s Explen uses slicing techniques to support efficient navigation through large models [10], reducing cognitive load by extracting relevant submodels and enabling focused analysis of specific concerns within extensive model repositories. While these works are fundamental for visualizing models and metamodels, especially those of considerable size, they do not provide solutions for defining a concrete syntax for creating models of a metamodel-defined language.

3.3 Usability of Model-Driven User Interfaces

Abrahão et al.'s work explores whether model-driven architecture can improve the usability of user interfaces generated automatically from high-level models, as we now propose. While existing approaches support UI generation, they do not explicitly integrate usability engineering into the development process. By evaluating the usability of the final generated interfaces, the study shows that identified usability issues can be used to refine platform-independent and platform-specific models, supporting the idea that usability can be improved through model-based development [1]. Our work relies on generating a concrete syntax for the models, which has proven to produce languages with high usability. In fact, we also demonstrate this for a case study using our approach.

Aquino et al. evaluate the usability of multi-platform user interfaces generated using model-driven engineering techniques across different platforms and screen sizes. An experimental study compares desktop and web interfaces generated from the same models and used on standard, large, and small displays, measuring satisfaction, effectiveness, and efficiency. Results show that efficiency is higher on desktop platforms and larger screens, while satisfaction tends to decrease on smaller displays. The findings highlight the need to enhance model-driven engineering approaches to better account for platform and device characteristics in order to improve the usability of generated user interfaces [4]. Although the block-based language we generate can be used on "small" screens because it runs in a browser, its usability is very unlikely to be good, as it has not been designed for such environments. It has shown positive results in desktop-like environments, which was our goal.

3.4 Block-Based Programming Environments

Block-based programming environments such as Blockly [30], Scratch [36], and OpenBlocks [40] have demonstrated the power of visually structured interfaces to lower entry barriers for novice users. By replacing textual syntax with modular, snap-together blocks, these systems enhance accessibility and engagement, and have been successfully applied across domains including linear programming [18], spreadsheets [34], industrial robotics [48], and Java programming [5]. We build on these same ideas and principles to create a generic solution for metamodel-based languages. The case study we present in Section 7 indeed demonstrates promising results.

3.5 Automated Visual Environment and Interface Generation

Recent work explores automated generation of interactive visual environments and user interfaces through declarative specifications, learning-based synthesis, and large-scale datasets. Transformer-based architectures have been proposed to translate visual designs into functional code, reducing the manual gap between UI mock-ups and implementation artifacts [17]. Complementing this direction, recent research introduces automated pipelines for large-scale user interface dataset generation, extracting structured layout and component information from web interfaces to support AI-enhanced design tools [44]. Large-scale multimodal datasets such as WebCode2M provide real-world webpage design–code pairs to support hierarchical UI code generation and benchmarking of design-to-code systems [31]. Such datasets significantly improve the scalability and robustness of automated UI synthesis approaches. Our approach builds on existing and proven interfaces, in particular block-based ones.

4 From Metamodel Elements to Blocks

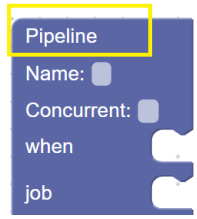
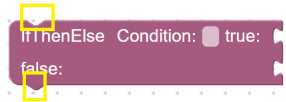
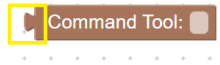
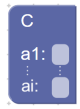
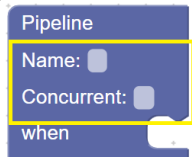
One of the main goals of our work is to generate visual modeling environments for existing model-based languages. The most common way to define such languages is through a metamodel [13]. Thus, in this section, we define how to map metamodel elements to blocks. It would also have been possible to adopt a more formal approach to the transformation by defining a dedicated metamodel for Blockly and then defining model-to-model transformations between each metamodel and the Blockly metamodel. However, no formal metamodel for Blockly currently exists. Moreover, developing such a metamodel within the scope of this work would likely introduce biases comparable to those arising from directly transforming the original metamodel into a Blockly representation.

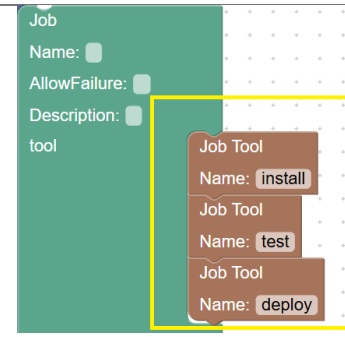
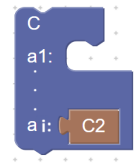
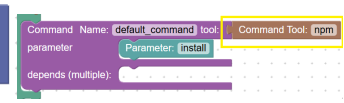
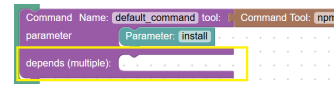
Arguably, EMF and its Ecore language form the most used environment for creating metamodels. Thus, we define this mapping for such kinds of metamodels. Nevertheless, the approach we propose can be applied to other modeling environments.

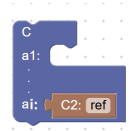
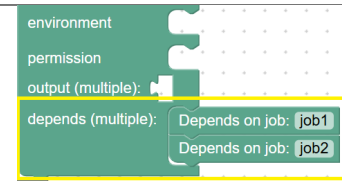
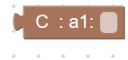
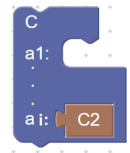
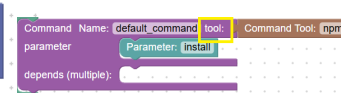
To translate Ecore metamodels into a block-based representation, we first extract the model's relevant structural elements, including class names, attributes, relationships, and containment references. Using this information, we systematically generate the corresponding blocks, with each block type derived directly from its associated class name.

Briefly, each class becomes a block where the class name is used as the block's label, while class attributes are represented as field inputs. Containment and relationship references are mapped to input fields or input statements, depending on their cardinality (e.g., one-to-many relationships are represented using statement inputs). Finally, we define appropriate connection rules, including next and previous statements, that reflect the model's structural semantics. Table 1 presents in detail the mapping between each Ecore element and the corresponding block feature. In the first column, we present each supported Ecore element. Currently, we support a subset of all elements, but, as we shall see in Section 6, it is sufficient to cover a significant amount of metamodels. The second column details how each element is mapped to a block feature. The third column presents the visual constructor that is created. To illustrate this process, in the last column we provide an example of a concrete block for an element of the metamodel presented in Figure 3.

Table 1. Ecore metamodel to Blockly mapping.

Ecore Feature	Mapping Details	Block	Example Block
EClass	An EClass, denoted *c*, is mapped to a block with identifier *c*. This block interacts with other blocks by receiving arguments as inputs and representing relationships through explicit connections. In our example, the *Pipeline* class in the metamodel is mapped to a block named *Pipeline*.		
EClass with EReferences to itself	An EClass named *c* with containment or reference relationships is represented as a vertically stackable block capable of containing child blocks. Both connection points are initially empty and are populated at runtime to represent relationships through block connections. In our example, the *IfThenClass* accepts references corresponding to the blocks that appear immediately before and after it on the canvas. In cases where the same object refers to itself, those validations would be normaly defined in the OCL domain, and therefore they would need to be checked at runtime.		
EClass without EReferences	An EClass named *c* with no EReferences is represented as a value block. This block can be plugged into the value inputs of other EClass blocks to indicate a reference to an instance of *c*. In our example, the *Command* block can be plugged into *Job* blocks.		
EAttribute	Each EAttribute a_i of an EClass *c* is mapped to an input field, such as a text field, list selector, or checkbox. The value entered in this field is read and processed at runtime. In our example, the attributes *Name* and *Concurrent* of the *Pipeline* class are mapped to a value box and a checkbox, respectively.		

Ecore Feature	Mapping Details	Block	Example Block
EReference (Containment, many=True)	A containment EReference between EClasses c (multiplicity 1) and c_1 (multiplicity [0..*] or [1..*]) is represented by allowing multiple c_1 blocks to be vertically stacked inside the c block. For multiplicity [1..*], adding a c_1 block automatically inserts one child block to enforce the minimum cardinality. These containment blocks are read and interpreted as containment relationships at runtime. In our example, multiple *Job Tool* blocks can be placed inside a *Job* block.		
EReference (Containment, many=False)	This case follows the same principle as the previous one, but restricts the number of children. For multiplicity [0..1], at most one child block can be added. For multiplicity [1..1], exactly one input is required. The containment relationship is evaluated at runtime. In our example exactly one *Command tool* block is placed inside a *Command*.		
EReference (Non-containment, many=True)	A non-containment EReference of an object c with multiplicity [0..*] or [1..*] is represented as a stacked input parameter within the c block. These references are resolved and read at runtime. In our example, a *Command* block can have several *Tool* block dependencies.		

Ecore Feature	Mapping Details	Block	Example Block
EReference (Non-containment, many=False)	A non-containment EReference of an object *c* with multiplicity [0..1] or [1..1] is represented as a single input value within the *c* block. This reference is read and resolved at runtime. In our example, the *Depends on job* block represents this non-containment reference.		
EAttribute. name	The name of an EAttribute a_i is used as the field identifier a_i, preserving its original case, and is displayed. In our example, the placeholder corresponds to the *command* text, which serves as the identifier for the attribute in the metamodel.		
EReference. name	The name of an EReference is used to identify the corresponding connection point. It is displayed in message0 using the placeholder %N. In our example, the placeholder corresponds to the *Tool* identifier.		

5 Design and Implementation of *Model2Block*

In this section, we present *Model2Block*, a command-line tool that implements the transformation defined in Section 4. We start by presenting its design (Section 5.1), followed by a description of how to generate the block-based language (Section 5.2), and then by a description of the generated modeling environment (Section 5.3).

We provide a reproducibility package that includes the tool's source code [2].

5.1 Design

Blockly [30] provides a framework for defining web-based environments for block-based constructions. Thus, our tool builds on Blockly to provide a modeling environment for metamodels.

To generate the language and the graphical interface, users interact with *Model2Block* by supplying an existing Ecore metamodel as input. The tool parses the provided metamodel and generates the components required by Blockly to create the corresponding web-based graphical interface, along with the blocks and rules that guide how they are put together.

Currently, *Model2Block* supports two output formats. First, it allows the generation of JSON Blockly configurations that can be imported into any Blockly environment. This corresponds to generating the blocks and the rules for their connection, forming a concrete syntax for models conforming to the metamodel. Thus, one can use the Blockly language definition produced by our tool to create their own modeling environment. Second, *Model2Block* allows the creation of a complete web interface incorporating the blocks generated by parsing the Ecore metamodel. This provides an out-of-the-box modeling environment, ready for deployment and usage.

The workflow of *Model2Block* is illustrated in the flowchart shown in Figure 4.

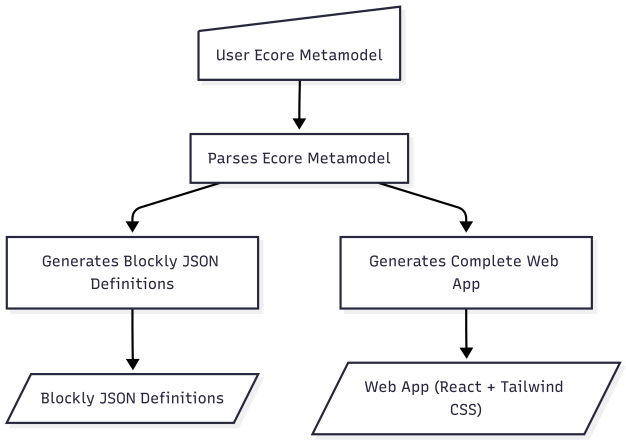


Fig. 4. Flowchart of *Model2Block*.

We have implemented *Model2Block* as a command-line interface (CLI), which provides a flexible, efficient mechanism for users to interact with the system by specifying input and output parameters via a variety of configurable options.

5.2 Generating Block-Based Languages

Regardless of whether the user generates only the JSON definition or the complete environment, our tool always produces the blocks' definitions based on the rules presented in Section 4. This process entails defining each block and the rules governing how they connect to one another, mirroring the semantics defined in the metamodel. This provides a concrete syntax for the language defined by the metamodel, enabling language users to create models that conform to the metamodel.

The transformation rules are specified declaratively and applied iteratively to the input metamodel. Each rule is triggered whenever a corresponding pattern is matched within the metamodel. The process begins by parsing the original metamodel to extract all relevant elements, including classes, attributes, containment relations, and associations. Based on this extracted information, we generate the corresponding Blockly blocks for each metamodel class and configure their associated fields, inputs, and parameters. Furthermore, we establish the required block connections (e.g., value and statement connections) to accurately reflect the structure and semantics defined in the metamodel. This means that the blocks can only be assembled in ways allowed by the metamodel. In fact, every model created using the generated environment conforms to the metamodel, that is, it is not possible to create incorrect models.

5.3 Generating Modeling Environments

The language generated by *Model2Block* can be integrated into a fully featured, web-based, block-based modeling environment derived from Ecore metamodels. As illustrated in Figure 3, this language constitutes only one component of the overall environment. The generated interface remains compliant with the original metamodel and is designed to streamline the model development process.

The language can be generated either as a standalone JSON-based Blockly component or as part of the complete modeling interface. In both cases, the language is generated and compiled in accordance with the metamodel, ensuring seamless interaction with the generated environment.

The generated application is distributed as a compressed archive that can be extracted, configured with the required dependencies, and executed as a standalone web application. *Model2Block* is implemented with React[1] and styled with Tailwind CSS[2], providing a flexible, customizable interface for developers and end users alike.

Key features of the generated environment include:

Modular Language Blocks Users interact with a visual drag-and-drop canvas to construct, modify, and visualize models. As shown in Figure 3.A, a palette displays all language blocks derived from the metamodel. These blocks can be dragged into the modeling workspace (Figure 3.B) to assemble structurally conformant models. When dragging and dropping the blocks, the rules derived from the metamodel, and in particular from the relations between the classes, determine how the blocks are mounted together. The blocks have visual clues that help users determine where each block can fit. For instance, the block Job has a small curve on its top edge. This shows that this curve must be placed in another block containing the opposite curve, which is the case of the "input" ports of the Pipeline block. Although there are several places in the Pipeline block where a Job seems to fit, the name of each "input" provides another clue for the user. In this case, there is a *job* "input" where the Job block should be dropped. In fact, Job does not fit anywhere else, as our tool generates a Blockly configuration to force it. When dropping the block, the environment exerts a "magnetic" force that helps the user place it correctly when it is close. As discussed in Section 3, Blockly has been successfully used in many other contexts.

Persistent Model Management The tool includes dedicated controls (Figure 3.C) that allow users to save and reload models. Models are persisted in EMF's XML format [46], allowing their usage in any other tool of the EMF context.

Text-Based Representation In addition to the graphical interface, a YAML-based tree view of the model is provided (Figure 3.D). We chose a tree-based view (although textual) as it is quite common for representing models (e.g., within the EMF environment [46]). This provides a canonical view for all models, allowing users to see a representation that may be more familiar.

Responsive Design The interface is fully responsive, adapting across desktop, tablet, and mobile platforms. This feature is quite relevant as it is often challenging to define visual languages that can be used on smaller screens.

6 Applicability Study

To demonstrate the capabilities of *Model2Block*, we conducted an applicability study using an independently created dataset of Ecore metamodels to evaluate its effectiveness [9]. Some of the metamodels contained in the dataset can be seen in Appendix A. This dataset was originally introduced in a previous study that proposed a methodology for extracting architectural information from model-driven projects to identify and analyze subsequent metamodel versions throughout their evolution [8]. The original authors validated their approach on a diverse collection of metamodels of varying sizes and application domains.

For our evaluation, we adopted the same dataset, which contains 53 curated metamodels organized into five domain clusters to ensure variability in both size and domain. The metamodels range

[1] https://reactjs.org
[2] https://tailwindcss.com

in size from 3 to 190 elements, measured as the total number of classes, structural features, and packages. Most of these metamodels are relatively small (1–50 elements), primarily originating from task management, Petri net, and GWPN domains. A smaller subset represents medium-sized metamodels (51–150 elements), while a few larger ones correspond to GMF metamodels. The dataset is openly available, ensuring the transparency and reproducibility of our evaluation.

During validation, we used *Model2Block* to automatically convert each metamodel into the equivalent block-based view. The conversion failed in three cases due to malformed files, which we were unable to parse. For the remaining 50 metamodels, we automatically performed a structural comparison between the original metamodels and their generated Blockly specifications. Specifically, we compared the parsed *classes*, *references*, *containments*, and *attributes* to verify structural consistency.

After applying *Model2Block* and manually inspecting the generated blocks and environments, the results show 100% accuracy in all successfully processed files. That is, all the Blockly environments generated correspond to the underlying metamodel, and the blocks and their relationship correspond to the metamodel specifications.

The resulting generated modeling environments can be found in our reproducibility package [2].

7 Case Study

We conducted a controlled user study to evaluate the usability and perceived workload of the block-based CI/CD environment generated with *Model2Block*, compared to native GitHub Actions[3], the most used tool for CI/CD [29]. The study aimed to determine whether a visual, block-based approach improves usability and reduces the effort required to author CI/CD pipelines. The application used in this study was generated using *Model2Block* and provides a block-based language derived from the metamodel presented in Figure 1. It generates the corresponding GitHub Actions specification in real time, performs validation, and supports executing the generated code. Although this work is part of a previously published work[4], for completeness, we present its summary in this section.

7.1 Method

The study followed a within-subjects design. Each participant authored equivalent CI/CD pipelines using both tools, covering common build, test, and deploy scenarios adapted from standard CI/CD examples. The order of tools and tasks was counterbalanced to mitigate learning effects. After completing the tasks with each tool, participants completed the System Usability Scale (SUS) questionnaire [14] and the NASA Task Load Index (NASA-TLX) questionnaire [32].

The study involved ten voluntary participants, all Master's students in Computer Science in the latter half of their first semester. Most participants reported limited prior experience with CI/CD tools. Written informed consent was obtained prior to participation. While the number of participants is limited and their profiles are relatively homogeneous, it reflects novice or early-stage users, for whom accessibility is particularly relevant.

7.2 Results

The block-based environment achieved significantly higher usability scores than GitHub Actions. The average SUS score was 82.2 ($SD = 10.9$) for the generated environment, compared to 51.0 ($SD = 19.9$) for GitHub Actions, yielding a mean difference of 31.2 points ($p < 0.001$, Cohen's $d = 1.98$). According to standard SUS interpretations, the block-based environment was generally rated as excellent, whereas GitHub Actions scores were mostly below average.

[3] https://github.com/features/actions

[4] We omit the reference as it would disclose our identities.

Perceived workload, measured using NASA-TLX, was lower for the block-based environment, with a mean score of 3.9 ($SD = 2.95$), compared to 7.65 ($SD = 4.98$) for GitHub Actions. Although this difference narrowly missed conventional statistical significance ($p = 0.056$), the effect size was large (Cohen's $d = -0.88$), indicating a practically meaningful reduction in cognitive effort.

7.3 Error Analysis and Discussion

Given CI/CD scripts are usually defined using text files (often YAML files) and that there is little support during their definition, , which is the case of GitHub Actions, their creation is error-prone. Thus, evaluating the number of errors of our approach compared to an existing one provides important evidence about the quality of the solution.

Overall error rates were comparable across both tools, but with different error patterns. GitHub Actions errors were primarily syntactic, such as malformed YAML or invalid keys, whereas errors in the block-based environment were mainly related to incorrect parameter configuration or block placement. These issues could be further mitigated through enhanced validation and contextual feedback.

7.4 Limitations

The main limitations of the study are the small sample size and the similar background of participants, which limit generalizability. Moreover, the limited number of metamodels/environments used is also a limitation. Nevertheless, the consistency of the observed results and the large effect sizes indicate that the block-based approach provides usability benefits, particularly for users with limited prior CI/CD experience.

8 Current Limitations

Model2Block demonstrates a promising approach to generate block-based visual interfaces from Ecore metamodels, but it still presents some limitations.

Currently, the implementation supports only a single metamodel at a time and does not handle multiple metamodels, which may limit its applicability in more diverse scenarios. Nevertheless, our applicability study showed its successful execution in all the correct metamodels in a public dataset containing more than 50 samples.

The tool currently does not support OCL constraints. The relationships between blocks are enforced by the blocks that can be connected, but the model may still contain errors if constraints are defined using OCL.

We have not implemented any customization option. For instance, it is not possible to change the colors, names, or other settings of the blocks.

There is also no support for incrementality; this means that if a metamodel changes, a new set of blocks and the corresponding environment must be regenerated.

9 Conclusions

To answer RQ1, we have devised a method and developed a command-line tool that automatically generates block-based visual interfaces from metamodels, enabling users to visually construct models with blocks that accurately represent valid elements of the metamodel. This ensures that visual models conform to the original structure/metamodel, making the tool useful for intuitive and error-resistant model creation in a web environment.

This is a significant step forward in the creation of modeling environments, as existing tools require extensive manual effort and produce visual languages with several usability issues [6]. Our method produces block-based languages in a fully automated way, thereby reducing significant

effort for the user. Moreover, block-based languages have been shown to be quite usable and useful in several contexts [25, 34, 35].

Although, for now, our tool only supports Ecore metamodels, the method we defined can be applied to any modeling environment, including BESSEER [16] (for which we are already working) or MetaEdit+ [47].

To answer RQ2, we successfully applied our approach to a dataset containing 53 metamodels. Note that this dataset is public and was not defined by the authors. The tool could not parse three metamodels due to syntax errors, but for the remaining 50, it successfully generated visual environments containing the expected blocks and their relationships, thereby validating the practical applicability of our tool.

Finally, we answered RQ3 by drawing on prior work that validated a modeling environment generated using *Model2Block*. Our user study showed that the environment's usability was high, above 82 out of 100, and that the workload was considered low by participants. While the limited number of participants and metamodels used does not show that all generated environments have these characteristics, it provides initial empirical evidence of such a scenario.

Future work will aim to enhance the tool's functionality to address the identified limitations. We intend to introduce richer customization options for both block definitions and the overall interface layout, thereby enhancing the tool's adaptability to diverse modeling domains and user requirements. Moreover, incrementally will be supported to allow a generated visual environment, its blocks, and the models created using the environment to evolve if the original metamodel evolves too.

Acknowledgments

This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2025 (https://doi.org/10.54499/UID/50014/2025).

The first author was supported by the doctoral grant 2022.13084.BD, financed by the Portuguese Foundation for Science and Technology (FCT).

References

[1] Silvia Abrahão, Emilio Iborra, and Jean Vanderdonckt. 2008. *Usability Evaluation of User Interfaces Generated with a Model-Driven Architecture Tool.* Springer London, London, 3–32. doi:10.1007/978-1-84628-941-5_1

[2] Author Anonymous. 2025. *Automatically Generating Visual Modeling Environments.* doi:10.5281/zenodo.17438001

[3] Muhammad Waseem Anwar and Federico Ciccozzi. 2022. Blended Metamodeling for Seamless Development of Domain-Specific Modeling Languages across Multiple Workbenches. In *2022 IEEE International Systems Conference (SysCon).* IEEE, Piscataway, NJ, USA, 1–7. doi:10.1109/syscon53536.2022.9773924

[4] Nathalie Aquino, Jean Vanderdonckt, Nelly Condori-Fernández, Óscar Dieste, and Óscar Pastor. 2010. Usability evaluation of multi-device/platform user interfaces generated by model-driven engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano-Bozen, Italy) *(ESEM '10).* Association for Computing Machinery, New York, NY, USA, Article 30, 10 pages. doi:10.1145/1852786.1852826

[5] Aryobarzan Atashpendar and Steffen Rothkugel. 2024. Block-Based Programming for Mobile with Conventional Exceptions and Automatic Evaluation. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024).* Association for Computing Machinery, New York, NY, USA, 597–603. doi:10.1145/3649217.3653549

[6] Ankica Barišić, Dominique Blouin, Vasco Amaral, and Miguel Goulão. 2017. A requirements engineering approach for usability-driven DSL development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering.* Association for Computing Machinery, New York, NY, USA, 115–128.

[7] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: blocks and beyond. *Commun. ACM* 60, 6 (May 2017), 72–80. doi:10.1145/3015455

[8] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2020. Detecting Metamodel Evolutions in Repositories of Model-Driven Projects. *J. Object Technol.* 19, 2 (2020), 14:1–22. doi:10.5381/JOT.2020.19.2.A14

[9] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2020. metamodelsdataset-ECMFA2020: Dataset for "Detecting Metamodel Evolutions in Repositories of Model-Driven Projects". https://github.com/gssi/metamodelsdataset-ECMFA2020. Associated with the ECMFA 2020 paper "Detecting Metamodel Evolutions in Repositories of Model-Driven Projects".

[10] Arnaud Blouin, Naouel Moha, Benoit Baudry, and Houari Sahraoui. 2014. Slicing-Based Techniques for Visualizing Large Metamodels. In *IEEE Working Conference on Software Visualization*. IEEE, Piscataway, NJ, USA, 25–29. doi:10.1109/VISSOFT.2014.14

[11] Paolo Bottoni and Antonio Grau. 2004. A Suite of Metamodels as a Basis for a Classification of Visual Languages. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. IEEE, Piscataway, NJ, USA, 83–90. doi:10.1109/VLHCC.2004.5

[12] Paolo Bottoni, Esther Guerra, Juan de Lara, et al. 2006. Metamodel-based definition of interaction with visual environments.. In *MDDAUI@ MoDELS*. CEUR-WS.org, Genova, Italy, 1–4. https://ceur-ws.org/Vol-214/paper10.pdf

[13] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice* (2 ed.). Morgan & Claypool Publishers, San Rafael, CA. doi:10.2200/S00718ED1V01Y201701SWE003

[14] John Brooke. 1996. *SUS: A quick and dirty usability scale.* Technical Report TR-38. Digital Equipment Corporation, Reading, UK. Originally published in 1986.

[15] Antonio Bucchiarone, Juri Di Rocco, Damiano Di Vincenzo, and Alfonso Pierantonio. 2025. Modeling in Jjodel: Bridging Complexity and Usability in Model-Driven Engineering. arXiv:2502.09146 [cs.SE] https://arxiv.org/abs/2502.09146

[16] Jordi Cabot. 2024. *The Low-Code Handbook: Learn How to Unlock Faster and Better Software Development with Low-Code Solutions.* Independently published, n.p.

[17] Tommaso Calò and Luigi De Russis. 2025. Advancing Code Generation from Visual Designs through Transformer-Based Architectures and Specialized Datasets. *Proc. ACM Hum.-Comput. Interact.* 9, 4, Article EICS013 (June 2025), 37 pages. doi:10.1145/3734190

[18] Hugo Da Gião, Jácome Cunha, and Rui Pereira. 2021. Linear Programming Meets Block-based Languages. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Piscataway, NJ, USA, 1–3. doi:10.1109/VL/HCC51201.2021.9576449

[19] Péter Domokos and Dániel Varró. 2002. An Open Visualization Framework for Metamodel-Based Modeling Languages. In *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, The Netherlands, 69–78. doi:10.1016/S1571-0661(05)80531-6

[20] Dirk Draheim, Melanie Himsl, Daniel Jabornig, Jürgen Küng, Werner Leithner, Peter Regner, and Thomas Wiesinger. 2010. Concept and pragmatics of an intuitive visualization-oriented metamodeling tool. *J. Vis. Lang. Comput.* 21 (2010), 157–170. doi:10.1016/j.jvlc.2010.03.002

[21] Dirk Draheim, Melanie Himsl, Daniel Jabornig, Werner Leithner, Peter Regner, and Thomas Wiesinger. 2009. Intuitive Visualization-Oriented Metamodeling. In *Enterprise, Business-Process and Information Systems Modeling*. Springer, Cham, Switzerland, 727–734. doi:10.1007/978-3-642-03573-9_60

[22] Eclipse Foundation. 2006. Eclipse Graphical Modeling Framework (GMF). https://www.eclipse.org/modeling/gmp/.

[23] Eclipse Foundation. 2013. Eclipse Sirius. https://eclipse.dev/sirius/.

[24] Eclipse Foundation. 2025. Sirius User Documentation. https://wiki.eclipse.org/Sirius/UserDocumentation.

[25] Neil Fraser. 2013. Blockly: A visual programming editor. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Piscataway, NJ, USA, 1–6. https://developers.google.com/blockly Google Blockly Project.

[26] Antonio Garmendía, Manuel Wimmer, Esther Guerra, Elena Gómez-Martínez, and Juan de Lara. 2020. Automated variability injection for graphical modelling languages. In *GPCE '20: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Association for Computing Machinery, New York, NY, USA, 15–21. doi:10.1145/3425898.3426957

[27] Sebastian Gebhardt, Sebastian Pick, Bernd Hentschel, and Torsten Kuhlen. 2018. Interactive Visual Analysis of Multi-dimensional Metamodels. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV 2018)*. The Eurographics Association, Brno, Czech Republic, 79–89. doi:10.2312/pgv.20181098

[28] Hugo Gião, Jácome Cunha, Rui Pereira, André Flores, Vasco Amaral, Gregor Engels, and Stefan Sauer. 2025. A Metamodel for Reengineering CI/CD Pipelines. Preprint.

[29] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2022. On the rise and fall of CI services in GitHub. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 662–672. doi:10.1109/SANER53432.2022.00084

[30] Google. 2012. Blockly: A Visual Programming Editor. https://developers.google.com/blockly.

[31] Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Yi Su, Bohua Chen, Dongping Chen, Xing Zhou, and Wenbin Jiang. 2025. WebCode2M: A Real-World Dataset for Code Generation from Webpage Designs. In *Proceedings of the ACM Web Conference 2025 (WWW '25)*. ACM, New York, NY, USA, 1834–1845. doi:10.1145/3696410.3714889

[32] Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Advances in Psychology*, Peter A. Hancock and Najmedin Meshkati (Eds.), Vol. 52. North-Holland, Amsterdam, The Netherlands, 139–183. doi:10.1016/S0166-4115(08)62386-9

[33] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, Boston, MA, USA.

[34] Bas Jansen and Felienne Hermans. 2019. XLBlocks: a Block-based Formula Editor for Spreadsheet Formulas. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, Piscataway, NJ, USA, 55–63. doi:10.1109/VLHCC.2019.8818748

[35] Sarah Claudia Krings, Enes Yigitbas, Kai Biermeier, and Gregor Engels. 2022. Design and Evaluation of AR-Assisted End-User Robot Path Planning Strategies. In *Companion of the 2022 ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Sophia Antipolis, France) *(EICS '22 Companion).* Association for Computing Machinery, New York, NY, USA, 14–18. doi:10.1145/3531706.3536452

[36] MIT Media Lab. 2007. Scratch Programming Language. https://scratch.mit.edu/.

[37] Pedro Lumertz, Luciano Ribeiro, and Leticia Duarte. 2016. User interfaces metamodel based on graphs. *J. Vis. Lang. Comput.* 32 (2016), 1–34. doi:10.1016/j.jvlc.2015.10.026

[38] Víctor López-Jaquero, Elena Navarro, Francisco Montero Simarro, and Pablo González. 2013. Metamodels Infrastructure and Heuristics for Metamodel-Driven Multi-touch Interaction. In *Human-Computer Interaction – INTERACT 2013.* Springer, Cham, Switzerland, 210–227. doi:10.1007/978-3-642-40480-1_14

[39] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. 2023. Dandelion: A scalable, cloud-based graphical language workbench for industrial low-code development. *Journal of Computer Languages* 76 (2023), 101217. doi:10.1016/j.cola.2023.101217

[40] MIT. 2009. OpenBlocks Visual Programming Environment. https://openblocks.org/.

[41] Daniel 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. doi:10.1109/TSE.2009.67

[42] Obeo. 2014. Acceleo. https://eclipse.dev/acceleo/. Accessed: Oct. 1, 2024.

[43] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: Programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[44] Francesca Russo, Tommaso Calò, and Luigi De Russis. 2025. Bridging Web and Figma: Automating Large-Scale UI Dataset Generation for AI-Enhanced Design. In *Companion Proceedings of the 17th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS Companion '25).* ACM, New York, NY, USA, 13–20. doi:10.1145/3731406.3734974

[45] Alberto Rodrigues Da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* 20 (2015), 1–28. doi:10.1016/j.cl.2015.06.001

[46] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, n.p.

[47] Juha-Pekka Tolvanen and Matti Rossi. 2003. Metaedit+ defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* Association for Computing Machinery, New York, NY, USA, 92–93.

[48] David Weintrop, David C. Shepherd, Patrick Francis, and Diana Franklin. 2017. Blockly goes to work: Block-based programming for industrial robots. In *2017 IEEE Blocks and Beyond Workshop (B&B).* IEEE, Piscataway, NJ, USA, 29–36. doi:10.1109/BLOCKS.2017.8120406

[49] Yuxuan Zhang, Peng Wang, Yifan Wang, and Jian Chen. 2017. BlocklyIoT: A web-based visual programming editor for configuring sensors and actuators in the Internet of Things. In *2017 IEEE 3rd International Conference on Big Data Security on Cloud (BigDataSecurity).* IEEE, IEEE, Piscataway, NJ, USA, 75–80.

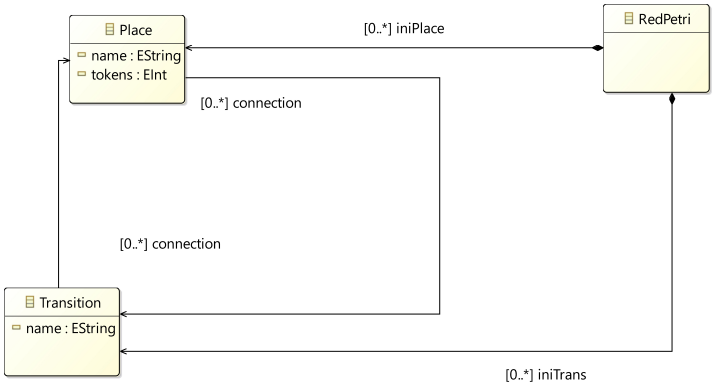
[50] Philipp Ziegler, Alexander Schummer, and Sandro Wartzack. 2012. Visualization of simulation-data-based metamodels during the product synthesis. In *Proceedings of NordDesign 2012.* Design Society, Aalborg, Denmark, 1–8. https://www.designsociety.org/publication/38594/Visualization+of+simulation-data-based+metamodels+during+the+product+synthesis NordDesign 2012, August 22–24, 2012.

[51] Steffen Zschaler, Will Barnett, Artur Boronat, Antonio Garcia-Dominguez, and Dimitris Kolovos. 2024. Move your MDE teaching online: The MDENet Education Platform. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems* (Linz, Austria) *(MODELS Companion '24).* Association for Computing Machinery, New York, NY, USA, 6–10. doi:10.1145/3652620.3687780

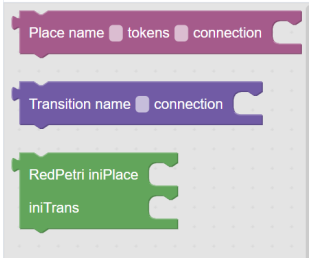
A Examples of Metamodels Used in the Applicability Study

In this section, we present four metamodel examples used in the applicability study presented in Section 6. We also present the blocks generated for each metamodel using our tool.

A.1 Petri Net



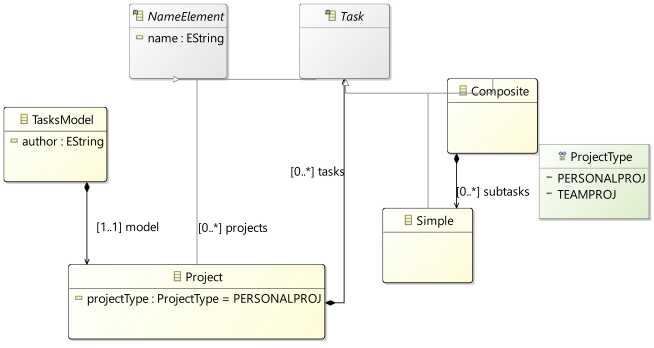
(a) Petri net metamodel, one of the metamodels in the dataset used for our validation.



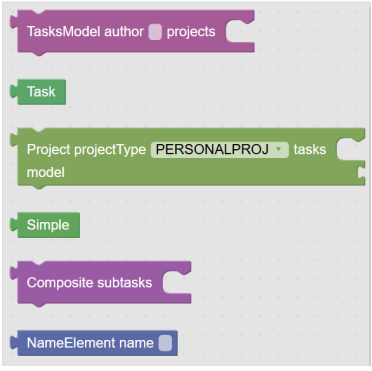
(b) Blocks generated from the Petri net metamodel.

Fig. 5. Petri net example.

A.2 Task Management



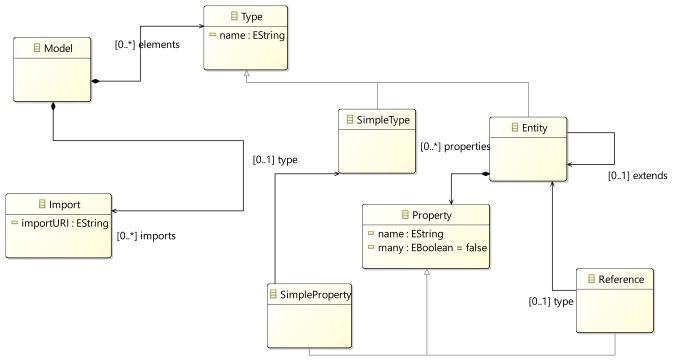
(a) Task management metamodel, one of the metamodels in the dataset used for our validation.



(b) Blocks generated from the task management metamodel.

Fig. 6. Task management example.

A.3 Entities

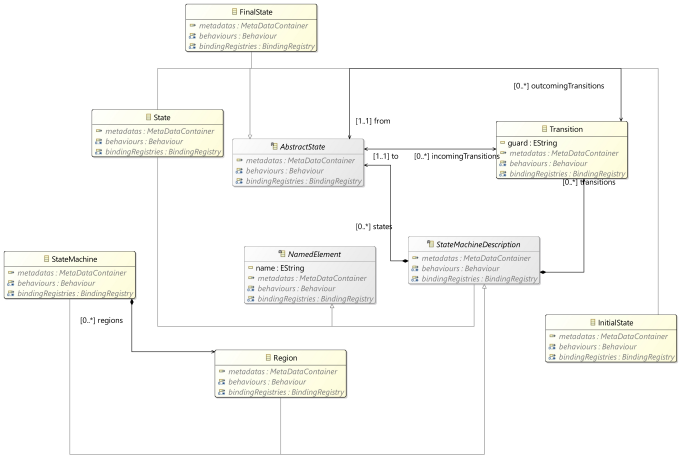


(a) Entities metamodel, one of the metamodels in the dataset used for our validation.

(b) Blocks generated from the entities metamodel.

Fig. 7. Entities example.

A.4 State Machine



(a) State machine metamodel, one of the metamodels in the dataset used for our validation.

(b) Blocks generated from the state machine metamodel.

Fig. 8. State machine example.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009