Electronic Communications of the EASST
Volume 84 Year 2025

12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation / 2nd AISoLA - Doctoral Symposium, 2024

Edited by: Sven Jörges, Salim Saay, Steven Smyth

Accelerating Graphical DSL Development: Live Metamodeling in Cinco Cloud

Daniel Sami Mitwalli

DOI: 10.14279/eceasst.v84.2678

Accelerating Graphical DSL Development: Live Metamodeling in Cinco Cloud

Daniel Sami Mitwalli [1 2 3]

University of Limerick, Ireland (mitwalli.daniel@ul.ie)[1]
CRT-AI: SFI Centre for Research Training in Artificial Intelligence[2]
Lero: The Irish Software Research Centre[3]

Abstract: Language engineering of graphical Domain-Specific Languages (DSLs) often involves inefficient repetitive compilations when using a generative approach for corresponding Integrated Modeling Environments (IMEs). Instead of generating and compiling dedicated IMEs for each language, this work proposes using a general-purpose IME to interpret the syntax and semantics of graphical DSLs. This approach eliminates the compilation time of dedicated IMEs and serves as a platform that streamlines the language engineering process, enabling real-time adjustments and direct use of the resulting graphical DSL.

Keywords: Domain specific languages, Model-driven engineering, Language-driven engineering, Cloud-based IDEs, Interpreter, Live metamodeling, Integrated Modeling Environments.

1 Introduction

Model-Driven Engineering (MDE) [Ken02, Sch06] focuses on the use of models to abstract domain-specific concepts to automate code generation, analyze and verify models, as well as streamline the process of software engineering. In MDE, DSLs [Fow10] serve as a tool for domain experts to express such models, being a special form of programming languages with an expressiveness tailored to a particular domain. Among DSLs, graphical DSLs [N+18] use graph representations to simplify modeling, either by visualizing information or transforming the models into a semantic representation that is further processed, e.g. as a remote procedure [Z+21], CI/CD pipelines [T+21] or even a full-stack web-application [B+16]. Such DSLs are typically supported by IMEs [S+24], which provide structured graphical editing capabilities. Fortunately, there are approaches that aid language engineering graphical DSLs and focus on automatically generating associated IMEs from the specification of such languages [N+18, G+24, B+21, MKK+14, E+13]. Cinco Cloud [B+22] was introduced as a platform to address this issue using a generative approach for web-based IMEs. It allows language engineers to specify graphical DSLs using textual meta-DSLs. From the specification of a language, an IME is generated and deployed for domain experts, ready to use. This method simplifies the creation of IMEs, but introduces a major inefficiency: every change to a DSL requires regenerating and recompiling the IME, leading to significant delays, potentially taking several minutes per iteration, creating substantial overhead in the language development workflow. While some approaches can update syntax changes almost instantly [MKK+14, B+21], adequate support for handling semantic modifications remains limited. It either requires recompiling and redeploying the IME [B+22], manually reloading/updating the editor [MKK+14, B+21], or is limited to a specific set of semantic operations [B+21].

This paper introduces a general-purpose IME for graphical DSLs that dynamically interprets both syntax and semantics at runtime. By eliminating the need for regeneration and recompilation, it significantly reduces development overhead. This, in turn, enables the immediate reflection of modifications to a DSL in the modeling environment without requiring restarts or long waiting times — an ability referred to as *live metamodeling*. It not only accelerates the language engineering but also makes DSLs development more efficient and responsive.

This paper is organized as follows. Section 2 introduces the foundational concepts of Models, MDE and DSLs, as well as IMEs, graphical DSLs and Cinco Cloud, setting the stage for this work. Section 3 presents the core ideas and challenges of interpreting graphical DSLs, focusing on the transition from a generative to an interpretive approach. Section 4 details an implementation of a general-purpose IME, the interpreter approach and its key concepts. Section 5 assesses the performance of the proposed approach by using an examplary graphical DSL, compares the results with related work, and depicts the practical implications. Finally, the Section 6 concludes the contributions and Section 7 outlines potential directions for future research and development.

2 Background

This section introduces the key concepts underlying this paper. It begins with models, MDE, and DSLs (Section 2.1), followed by a description of IMEs (Section 2.2) and their role in MDE. Finally, it introduces the foundation of this paper, graphical DSLs (Section 2.3) and Cinco Cloud (Section 2.4), a platform for creating such DSLs.

2.1 Models, MDE and DSLs

Models [N+18, Fow10, W+23] serve as abstractions of real-world systems, capturing key aspects of a specific domain in an abstract manner. They omit unnecessary details while retaining essential domain-relevant information. This way, models help manage complexity by structuring information in a way that is easier to understand and manipulate. MDE [Ken02, Sch06, W+23] leverages models as primary artifacts to streamline the software development process, incorporating automated code generation, model transformation, and validation. Models can be expressed using modeling languages that are specified by their structure, elements, and constraints. Such constraints describe the domain of valid models, which can be expressed by a model itself, i.e., a metamodel — a higher-level model that acts as a ”model of models”. The metamodel establishes the rules, elements, and relationships for creating valid models within a domain. Thus, the core of modeling languages can be expressed through a metamodel. Since modeling languages describe the structure of models, metamodeling languages describe how to construct metamodels themselves. Such metamodeling languages rely on a meta-metamodel to provide the foundational rules for defining modeling languages. A subset of modeling languages are DSLs. They are tailored to specific domains, providing precise and expressive ways to describe domain-specific problems. Unlike General-Purpose Languages (GPLs), which aim for broad applicability, DSLs focus on specific problem domains, making them more intuitive to domain experts. That way, DSLs can bridge the gap between technical and non-technical stakeholders, improving clarity and maintainability [Fow10, N$^+$69]. The syntax of a DSL consists of an abstract syntax, which defines the structural rules of the language, and a concrete syntax, which specifies its appearance, i.e., its textual or graphical representation [W$^+$23]. The semantics of a DSL determine the meaning of DSL expressions and determine how they are processed, transformed, or executed [Fow10]. Thus, DSL code can be used to drive code generation, model transformation, or direct interpretation, making them powerful tools for domain-specific automation. A DSL can also be used to describe the metamodels of other DSLs. Such a DSL is called a meta-DSL in this paper.

2.2 Integrated Modeling Environments

IMEs [S$^+$24] are dedicated workbenches for modeling languages, i.e. the creation, editing and management of models. They provide specialized tools and graphical or textual editors for the respective modeling languages with validation, transformation and code generation functions to enable model-driven engineering. By providing specialized tools, IMEs enable domain experts and developers to interact efficiently with models and reduce the complexity of model creation and editing. Similarly, workbenches that enable the creation of metamodels of modeling languages through metamodeling languages are called meta-IMEs in this paper.

2.3 Graphical DSLs

In contrast to classic textual DSLs, there are also graphical DSLs [N$^+$18]. Instead of a textual representation, such languages have a concrete syntax consisting of graphical visual representations. The main goal of graphical DSLs is to simplify the process of describing models by providing a visual interface instead of requiring the user to write textual code. To support such languages, graphical IMEs are used, which provide graphical modeling editors consisting of components such as a canvas, an element palette, and user interactions such as drag-and-drop functionality.

2.4 Cinco Cloud

Cinco Cloud [B$^+$22] is a holistic web-based language engineering environment that allows users to specify graphical DSLs, generate corresponding IMEs, and deploy and use them directly in a single web application. Cinco Cloud uses a Kubernetes[1] cluster to dynamically deploy personal workspaces with meta-IMEs for designing graphical DSLs or dedicated IMEs for specific languages, all with a few clicks. Both meta-IMEs and IMEs are custom editors built on the Theia Platform[2]. For the specification of graphical DSLs Cinco Cloud serves a meta-IME providing two textual DSLs[3]. One to specify the abstract and one to specify the concrete syntax of a graphical language:

[1] https://kubernetes.io/
[2] https://theia-ide.org/
[3] Previously, there was a third meta-DSL called Cinco Product Definition (CPD) for configuring the resulting IME, which is now obsolete [N$^+$18, B$^+$22].

- The Meta Graph Language (MGL) allows a language engineer to specify the abstract syntax of a graphical DSL. It allows the specification of graphical model elements based on hierarchical graph structures, i.e. nodes, edges, containers and graph models, as well as their attributes and interoperability constraints such as containment or edge relations. Graph models represent graphical DSLs and contains nodes and edges, where edges are the connections between two nodes. A container is a node that can contain other nodes. All these (abstract) types can also be generalized and specialized in a polymorphic way. All model element types can be associated with attributes based on predefined primitive types (such as string, boolean, and numeric types), user-defined enums, model element references, and user-defined types. In addition, all model elements and attributes can be annotated with additional functionality and semantics, e.g. to hide an attribute from the user interface or to associate an implemented generator with a specific graph model type.

- The Meta Style Language (MSL) allows a language engineer to specify the concrete syntax of a graphical DSL, i.e. the visual representations, using shapes such as ellipses, rectangles, polygons, images, lines, arrows and text. Visual representations can be specified using two main constructs: Styles and Appearances. Styles are structural representations and include decorators for edges, such as arrowheads or labels, or shapes for nodes, such as polygons or ellipses. MSL files can be imported from an MGL file and then styles for nodes and edges can be referenced. Appearances contain visual information, such as color or line size, that can be applied to the shapes and decorators of styles. This allows nodes and edges to be represented in a variety of ways, visually aligning a graphical DSL with a domain expert's language.

In Cinco Cloud, language engineers can add semantics to a graphical DSL by implementing an interface in a GPL and reference it via an annotation on the associated model element type. Such semantics are code generators, user interactions like double-clicks or context menu entries. Within a Cinco Cloud meta-IME, the two meta-DSLs can be used to create graphical DSL specifications from which a language engineer can generate a dedicated IME with the click of a button. This involves feeding the language specification into an IME generator [B+22] that generates the corresponding source code and uploads it to a build-job service within Cinco Cloud's Kubernetes cluster where it is compiled. Once the build-job is complete, a domain expert can select and deploy the IME via a web interface and finally start using the graphical DSL.

3 Interpreting Language Specifications of graphical DSLs

Cinco Cloud's generative approach to developing graphical DSLs involves an iterative process with multiple steps (see Figure 1). Language engineers define meta-DSL files, implement associated semantics (e.g., for code generators), and generate IME code with a single click. The code is automatically compiled in the backend. They then use the new IME to assess their design decisions, including the runtime behavior of the DSL and the semantic output of e.g. an implemented code generator. If issues arise, they adjust the meta-DSL files or semantics, and must repeat the process by recompiling the IME, a process required for each change. This repeated compilation accumulates significant time, considerably slowing the language engineering

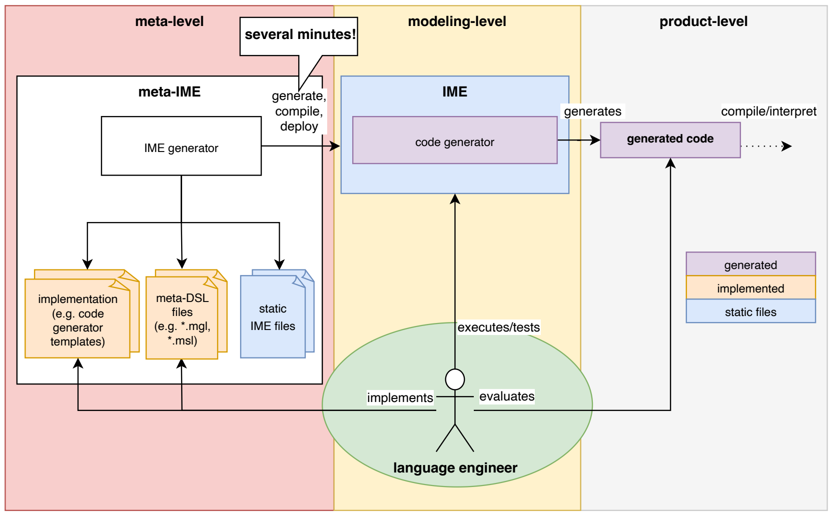


Figure 1: Designing, implementing and evaluating a graphical DSLs in Cinco Cloud.

process and making tasks like debugging highly expensive. As noted in [B+22], the generation and compilation can take several minutes before changes are reflected in the IME, highlighting the need to minimize this costly overhead.

In Cinco Cloud's generative approach (see Figure 2, A), the meta-IME provides language support[4] for the meta-DSLs to aid the language engineering process. Files written in the meta-DSLs are used as an input for an IME generator. The generator then produces source code for an IME dedicated to the specified graphical DSL(s). Most of the code of an IME typically contains a large static chunk of components written to integrate and use the expressive generated parts. A resulting dedicated IME must be recompiled for each change. To avoid these time-consuming compilation steps, a completely static, general-purpose IME can be constructed (see Figure 2, B). Since the information of the meta-DSLs files represents the specification of the language, which is used to create the generated components of a dedicated IME, this specification of the static components can theoretically also be used and interpreted directly. This is possible by replacing the generated components with a meta-DSL parser, and the resulting parser output is consumed directly by the static components. This approach eliminates the need to generate source code, and a resulting generic IME only needs to be compiled once. The meta-IME is similar to the IME, but has the additional task of providing language support for the meta-DSLs. The interplay between the meta-DSL parser and the static components of the general-purpose IME forms the basis of the interpreter approach presented in this paper. The goal of this approach is that the language engineer's modifications to a meta-DSL file by adding, deleting, or modifying an element are directly detected by the parser and propagated to the static components. This creates a responsive workflow that is referred to and proposed in this paper as *live metamodeling*.

[4] https://microsoft.github.io/language-server-protocol

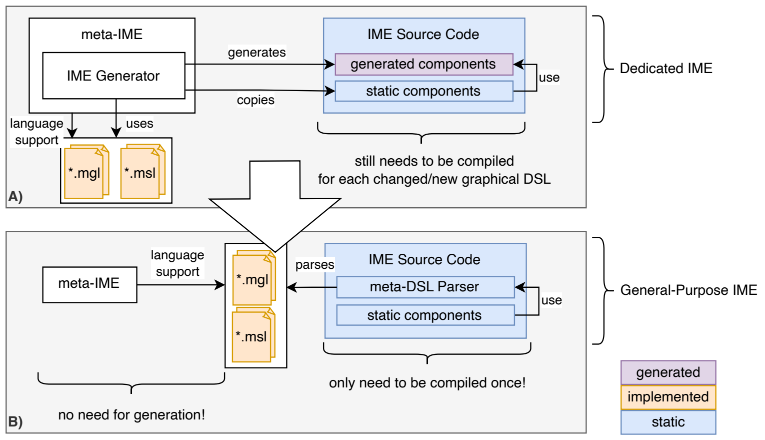


Figure 2: From a generative approach to an interpreting general-purpose IME that only needs to be compiled once.

3.1 The Structure of the Meta-Specification

In order to realize the interpretation appropriately, the parser transforms the input of the meta-DSL files into a suitable representation that can be used by the static components. Such a representation must describe all language specifications that can be described by the MGL and MSL, and is therefore called a meta-specification. Thus, all information that can be expressed by the meta-DSLs must be captured by the meta-specification in order to create instances from the specified types of the associated graphical DSLs. Since the MGL allows languages to be composed by importing and extending types using polymorphy from multiple languages into one, it means that the meta-specification needs to be able to contain compositions of languages. Because of such links between languages, a more specialized structure can be used to resolve such information before it is used by the static components. Therefore, a structure with a shallow type system based on the generalized types of the meta-DSLs is established, which is more sophisticated than a generic parse tree or Abstract Syntax Tree (AST).

3.1.1 The Framework of the Meta-Specification

Figure 3 shows the concept of the structure, which contains sets for all specifiable types of the MGL (see orange boxes), i.e., graphs, nodes, edges, and custom types, as well as all types of the MSL (see purple boxes). Note that this structure potentially contains all of the specified types of compositions of meta-DSL files together.

Both, graph and node types can contain multiple other nodes, i.e. a graph is, and a node can be, a container of model elements (see *ModelElementContainer*) and still be included in the corresponding set in the meta-specification (see *GraphTypes* and *NodeTypes*). Edge types are part of the set of edges (see *EdgeTypes*). The set of custom types includes user-defined types and enumerations (see *UserDefinedType* and *Enum*). Both are not considered modeling elements, but are still specified in the abstract syntax by the language engineer. All graphs, nodes, edges and user-

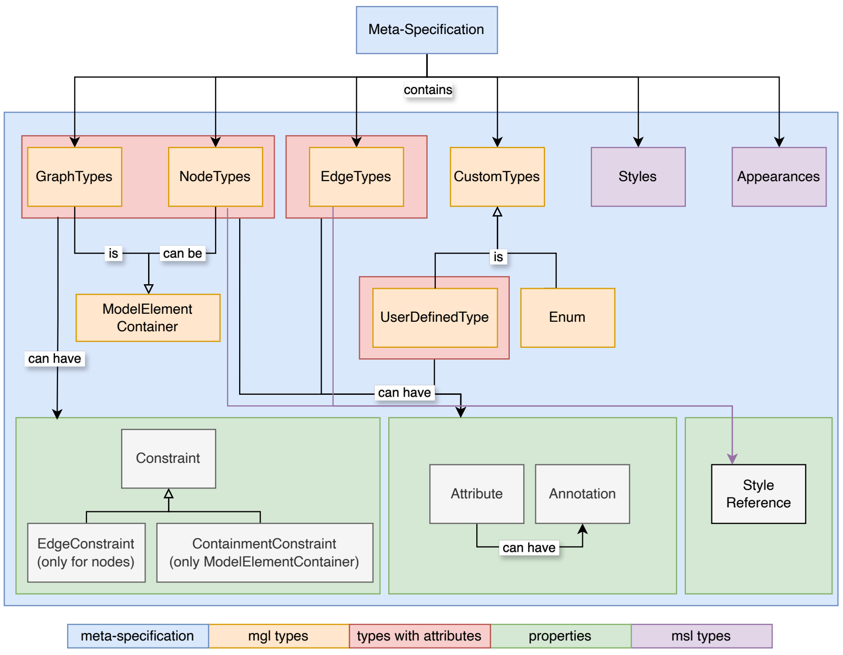
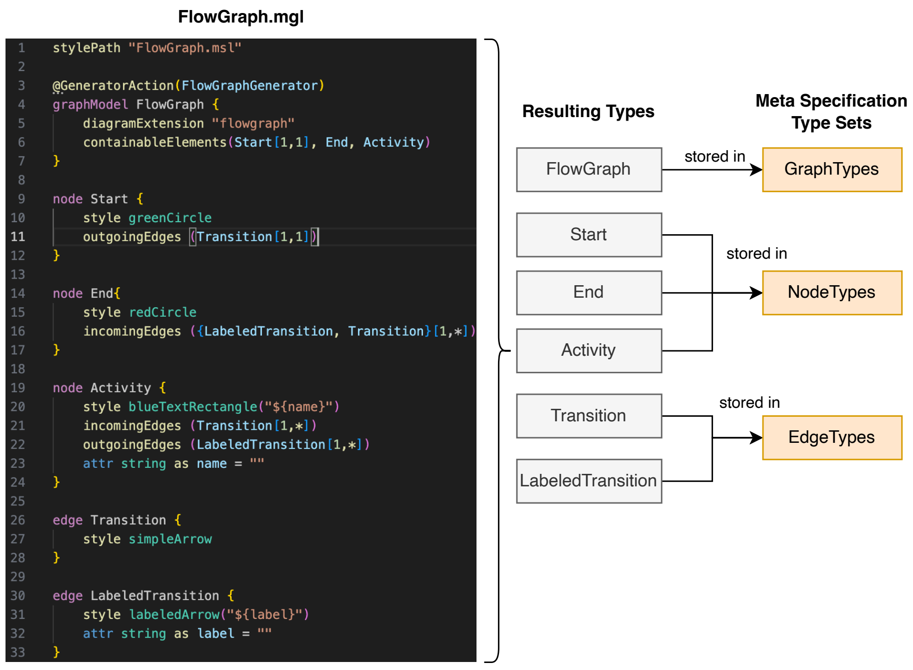


Figure 3: The structure of a meta-specification.

defined types (see red boxes) can be annotated (see green middle box) and contain attributes such as primitive or user-defined types. Attributes can also be annotated, e.g. if they need to be hidden from the user interface or handled in some other way. Graphs and nodes can be associated with specified containment constraints (see green left box). Graphs and containers can specify which (and how many) node types can be contained at runtime (see ContainmentConstraints), and each node type, whether if it is a model element container or not, can specify which types of edges can be connected to it, incoming and outgoing (see EdgeConstraints). All of these constraints play a key role, in specifying and configuring the behavioral model of the IME by governing the interoperation of model elements. Styles and appearances are contained in corresponding sets (see purple boxes), which serve as the source of all possible visual expressions of the concrete syntaxes at runtime. To realize a connection between the abstract node and edge types and their corresponding concrete style, both types have a specified reference property that points to a style via an identifier in the MGL [N+18].

To make this structure more comprehensible, an example of a graphical DSL is specified below using the MGL and MSL and how the resulting information is represented in the meta-specification (see Figure 4). In the MGL, a specified *graphmodel* type defines the structure of a graphical DSL. In this case, the *FlowGraph DSL* is specified in terms of its components and rules. Graphical models of the FlowGraph DSL are specified to have the file extension "*.flowgraph" (*diagramExtension*). The DSL includes three types of nodes, each with its own visual style and connection rules: *Start*, *End*, and *Activity*. A *Start* node is styled as a green circle and must appear exactly once in every model. It connects to one other node via a single edge of the specified *Transition* type. An *End* node is styled as a red circle and can have multiple incoming



```
stylePath "FlowGraph.msl"

@GeneratorAction(FlowGraphGenerator)
graphModel FlowGraph {
    diagramExtension "flowgraph"
    containableElements(Start[1,1], End, Activity)
}

node Start {
    style greenCircle
    outgoingEdges (Transition[1,1])
}

node End{
    style redCircle
    incomingEdges ({LabeledTransition, Transition}[1,*])
}

node Activity {
    style blueTextRectangle("${name}")
    incomingEdges (Transition[1,*])
    outgoingEdges (LabeledTransition[1,*])
    attr string as name = ""
}

edge Transition {
    style simpleArrow
}

edge LabeledTransition {
    style labeledArrow("${label}")
    attr string as label = ""
}
```

Figure 4: How the FlowGraph DSL types are stored in type sets of the meta-specification.

edges of the *Transition* or *LabeledTransition* type, and has no outgoing connections. An *Activity* node is represented as a blue rectangle that displays a *name* attribute. It requires at least one incoming *Transition* edge and one or more outgoing *LabeledTransition* edges. Two edge types define the flow between nodes: A *Transition*, styled as a simple arrow, represents a direct flow from one node to another. A *LabeledTransition*, shown as a labeled arrow, carries a string *label* to annotate the transition. Thus, the overall FlowGraph DSL allows the graphical modeling of structured, directional processes, where activities are linked by transitions and labels.

Several node, edge, and graph model types result from *FlowGraph.mgl* (see Figure 4, *Resulting Types*). These types are categorized and mapped into type sets of the meta-specification based on their typing, as determined by the MGL. These type sets are *GraphTypes*, *NodeTypes*, *EdgeTypes* and *CustomTypes* as mentioned in Figure 3. Any entity that can be specified in the MGL may have additional information, including attributes, annotations, references to a style, and constraints. This information is preserved in the entities to which the MGL-specified types are mapped within the meta-specification. Although not part of the FlowGraph example, when an element type is defined as a *container* in the MGL rather than a *node*, it is classified as a *NodeType* and receives an attribute identifying it as a *ModelElementContainerType* (see Figure 3). The MSL allows styles and appearances to be specified. The resulting entities are mapped similarly to those specified via the MGL (like in Figure 4) and are assigned to the corresponding *Style* and *Appearance* set shown in Figure 3.

3.1.2 Eagerly Resolving Inherited Properties

While the meta-specification maps and merges all the information described by meta-DSL files, the structure can also be used to prepare certain information before it is used by the static components of the general-purpose IME. Analogous to traditional programming languages, MGL supports inheritance and the specification of abstract entities. As mentioned above, such entities can be nodes, edges, graph models or user-defined types. Therefore, an element receives all the properties of its parents, i.e. attributes and constraints. As abstract entities are only relevant to the internal design of the language, they are not available to domain experts, e.g. in the palette of the IME. For this reason, their information can be applied directly to the concrete inheriting entities, and they themselves can be largely omitted within an IME (the origin and ancestry information can be stored separately, but this is out of scope). The inherited sets of properties can be resolved for each non-abstract entity in the meta-specification along its inheritance chain, but must be merged in a consistent manner in the case of *similar properties*:

- All attributes with the same identifier are overridden. An exception to this mechanism is when an inherited attribute shares the same identifier with a child's attribute, but not the type, which is considered an error.
- All constraints of the ancestors are collected. A constraint of an ancestor is overridden by a constraint of a descendant, iff they target a similar set of types, overridding the associated cardinalities.

As in traditional programming languages, the properties are collected from the most distant ancestor to and for the current entity, providing the most concrete blueprint for instantiation. The MGL supports model element inheritance not only within a single MGL file, but also supports the inheritance chains across multiple meta-DSL files, e.g. to implement nested languages. In addition, a constraint can target entities from other MGL files and entities that are ancestors of such. For example, consider a containment constraint that targets a node n. The constraint can also target all nodes that inherit from n, if there are any. This includes not only any inheriting node n_{local} from the same MGL file, but also any node n_{ext} from any external MGL file, that inherits from n (the same goes for edge constraints and their respective targeted edges) or even any node inheriting from n_{local} or n_{ext}. Cyclic inheritance between entities is ignored as it is considered an error. There are two ways to deal with this: resolve the properties in an *eager* or *lazy* manner. The approach presented in this paper, uses the eager way. The targeted entities of the constraints are resolved before the information of the MGL files is used by the components of the IME. All subtypes (and subtypes of subtypes) from all MGL files, that inherit from the targeted entities are collected and added to the set of targeted types of the constraint and all abstract targeted types are removed from the constraint. Thus, when an entity needs to be checked against the constraint, all concrete information is available. This approach is more appropriate and less computationally expensive than the lazy approach, which computes and checks whether an entity is a target of a constraint by resolving all related supertypes on demand.

As an example of eagerly resolving properties of the meta-specification see Figure 5. It depicts how specified nodes and containers that inherit properties from each other are resolved. The container *Cont2* inherits several properties from two containers, the abstract *ACont* and the

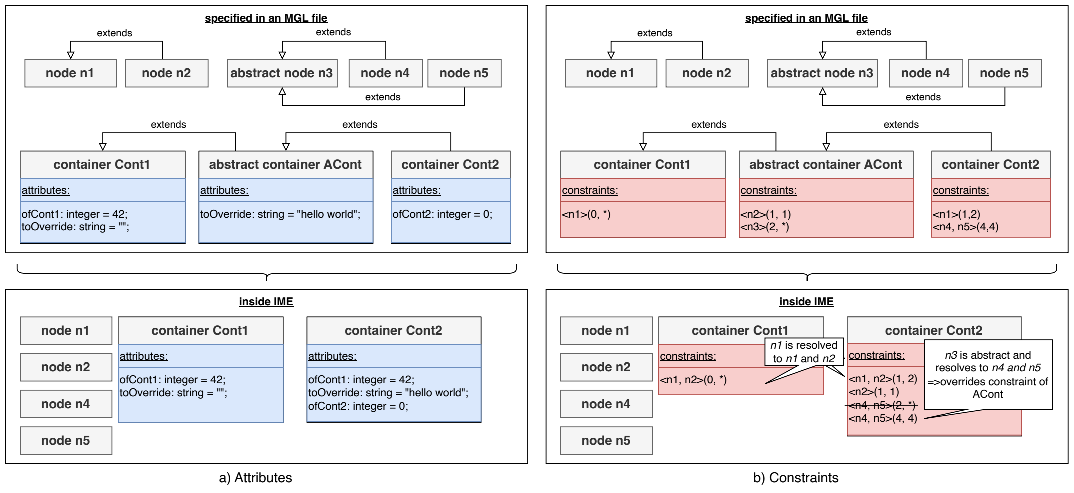

specified in an MGL file
extends
node n1
node n2
abstract node n3
node n4
node n5
container Cont1
abstract container ACont
container Cont2
attributes:
ofCont1: integer = 42; toOverride: string = "";
toOverride: string = "hello world";
ofCont2: integer = 0;
inside IME
ofCont1: integer = 42; toOverride: string = "hello world"; ofCont2: integer = 0;
a) Attributes
constraints:
<n1>(0, *)
<n2>(1, 1) <n3>(2, *)
<n1>(1,2) <n4, n5>(4,4)
n1 is resolved to n1 and n2
<n1, n2>(0, *)
n3 is abstract and resolves to n4 and n5 =>overrides constraint of ACont
<n1, n2>(1, 2) <n2>(1, 1) ~~<n4, n5>(2, *)~~ <n4, n5>(4, 4)
b) Constraints

Figure 5: Resolving properties of specified model elements in the IME.

concrete *Cont1*. Within the IME, there are only the containers *Cont1* and *Cont2* and the abstract *ACont* is omitted. To understand the notation of the figure, some details are explained below:

- Attributes are specified in the form $< name >:< type >=< defaultValue >$, where *name* is the name of an attribute and *type* is a primitive type such as string, boolean, and integer. The *defaultValue* specifies any value that the associated type allows.

- A constraint has the form $< e_1, e_2, ... > (lb, ub)$. Here e_i, with $i >= 0$, specifies a set of entities that specify the target of the constraint, i.e. which entities are part of the constraint. *lb* denotes the lower bound and *ub* the upper bound that can be associated with the targets of the constraints, i.e. the cardinality. E.g. given three nodes A, B, and C, and a container *Cont* with the constraints $< A, B > (0, *)$ and $< C > (1, 2)$. The container *Cont* can contain any number of nodes of types A and B (indicated by the wildcard upper bound $*$), but must contain at least one node of type C and a maximum of two.

Inherited information is passed on to *Cont2* (see *a*), such as the attribute *toOverride*, which is introduced by *Cont1* but overridden by *ACont*. This behaviour is analogous to the polymorphism of traditional programming languages. Constraints are handled in a similar way, but there are four possible effects due to potential collisions (see *b*):

- Resolving inheritance within constraints: The type *Cont1* is constrained to contain any number of nodes of the concrete type *n1*. Since *n2* is a descendant of *n1*, it is also of type *n1*. In the IME, this results in a resolved constraint that allows any number of *n1* and *n2* to be contained in *Cont1*.

- Constraint inheritance: The type *Cont2* inherits the condition $< n2 > (1, 1)$ from *ACont* without any side effects.

- Resolving inheritance within constraints using abstract types: *ACont* introduces a constraint $< n3 > (2,*)$ that requires it to contain at least two nodes of type *n3*. Since *n3* is an abstract node, the constraint is resolved so that it affects all its descendant types, i.e. *n4* and *n5*, while *n3* is omitted. This results in $< n4, n5 > (2,*)$. This rule does not exist in the IME, as it is overridden by the final effect.

- Override resolved inherited constraints: Resolving constraints and inheriting constraints can lead to a collision. Cont2 introduces the constraint $< n4, n5 > (4,4)$, but inherits the constraint $< n4, n5 > (2,*)$. As the introduced $< n4, n5 > (4,4)$ is more recent, it is kept and $< n4, n5 > (2,*)$ overridden.

3.2 Interpreting Semantics of graphical DSLs

The MGL supports various forms of semantics that can be associated with model elements such as user interactions (hooks and actions), code generators, interpreters and validators [N+18]. Model element types are associated through the use of annotations. The name of an annotation determines the type of semantics, while the value is typically a series of strings and contains a referencing identifier for the code implementation, as well as static configuration parameters for code execution. Semantics are implemented using a GPL and the implementation is based on interfaces provided by the IME runtime Application Programming Interface (API) and framework. In Cinco Cloud, the associated files are compiled with other sources of the IME after implementation. The approach proposed in this paper moves away from the compilation for language engineering to reduce the associated overhead. In order to interpret the semantics of a graphical DSL at runtime, this approach proposes to integrate an existing GPL interpreter into the general-purpose IME. During language engineering, all language files can be created, implemented and executed directly in the general-purpose IME, which provides a framework and interfaces for any kind of semantics, such as generators and validators. The code for these semantics is read and interpreted at runtime. This way, the general-purpose IME allows the language engineer to see the semantic changes and effects of the language being designed directly in the IME at runtime, providing the same flexibility in developing the semantics as in developing the syntax.

Figure 4 shows an example of the *FlowGraph* model type annotated with *@GeneratorAction* indicating an association with a generator. In Cinco Cloud, this annotation creates a button in the GUI for FlowGraph models that are associated with a specific generator. In this example, the generator is implemented as the *FlowGraphGenerator* class in a GPL (here TypeScript[5] or JavaScript) by the language designer and extends an interface provided by Cinco Cloud's API. Rather than integrating or compiling the implemented class into the application, an interpreter is executed when a generator button is clicked. This interpreter reads in the API and other necessary contextual information, as well as the implemented class. It then executes the associated generator using the current state of the model as an input.

[5] https://www.typescriptlang.org/

4 Implementation of a General-Purpose IME

The proposed concepts around the general-purpose IME are implemented for Cinco Cloud[6], replacing its previous generative approach described in [B+22]. The implementation is based on a Docker image[7] (see Figure 6). Docker provides a runtime environment for running isolated applications independent of the underlying infrastructure, ensuring consistency across different environments. The Docker image provides a Theia IDE based on the Theia Platform[8], a framework for building web-based editors, which forms the static foundation of both the general-purpose IME and the meta-IME. The general-purpose IME consists of two main components, most of which are implemented in TypeScript using NodeJS[9]:

- The client is a NodeJS module embedded in a Theia extension (see *IME Theia Extension*). The front-end of the client is largely based on the Graphical Language Server Platform (GLSP)[10] to provide various front-end components for the user of a graphical language, such as a modeling canvas, a palette to create the specified model element types and the possible user interactions. When the front-end is initialized, the client connects to the server and fetches the meta-specification. Conversely, when the server updates the meta specification, the latest version is passed from the server to the client so that the client is always up to date.

- The server is a NodeJS standalone application (see *Standalone Server*). It is executed using a Theia extension once the Theia backend is running (see *1*). The IME server accepts several command-line interface (CLI) parameters at startup, such as the port and the Uniform Resource Identifier (URI) path to the server, as well as the location of the workspace folder, the language folder, and the execution modes (i.e., language design mode and language user mode described in the next subsection). When the server is started, the language specifications are loaded and integrated into the server (see *2*), using Langium[11] to parse the meta-DSL files into ASTs, and transform them into the structure of the meta-specification. The meta-specification is then kept in-memory, ready to be propagated to the client (see *3*), and used by the static components of the IME.

Communication between the client's model editor and the server is realized using the GLSP. It was invented for the development of diagram editors and provides a Remote Procedure Call (RPC)-based [B+84] protocol for handling graphical models with actions, as well as a framework that includes corresponding action handlers and front-end components such as a modeling canvas and a palette. The only difference between the meta-IME and a general-purpose IME used for modeling only is an additional Theia extension that provides language support for the meta DSLs (see *Language Server Theia Extension*).

6 An open source implementation can be found at: https://gitlab.com/scce/cinco-cloud
7 https://www.docker.com/
8 https://theia-ide.org/
9 https://nodejs.org/
10 https://eclipse.dev/glsp/
11 https://langium.org

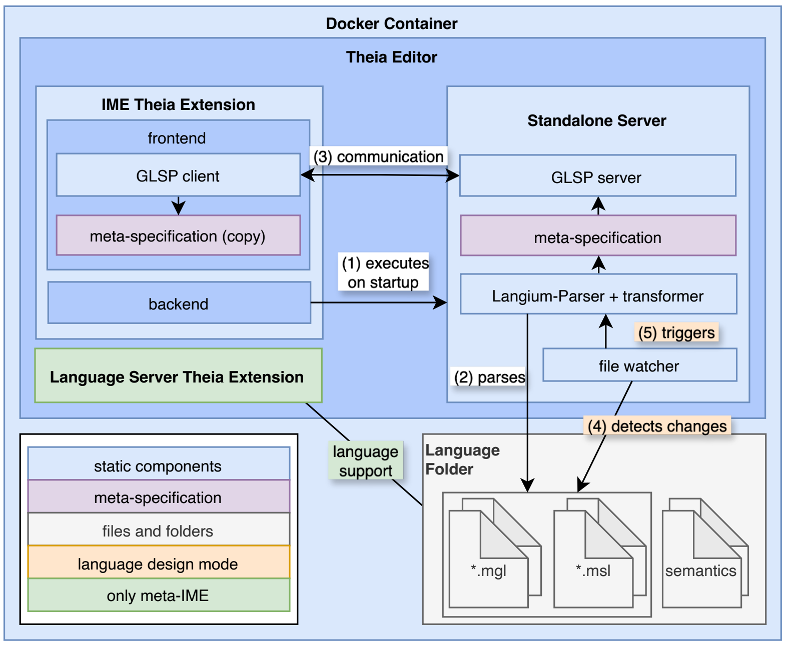


Figure 6: The implemented general-purpose IME, its startup and the change detection on meta-DSL files.

4.1 Handling Model and Language Files

The general-purpose IME handles several types of files. The meta-DSL files, the semantic files created by the language engineer, and the model files associated with the graphical DSL being created. A language engineer modifies language files, such as semantics and meta-DSL files, while the domain expert is not allowed to do so. To manage a set of these language files, a folder containing them is used, called the *language folder*. The folder containing the model files is called the *workspace folder*. Since a domain expert should not modify a language, but use it, a language folder must be protected from access within the IME when a domain expert uses it. To achieve this distinction, the implementation uses two execution modes:

- *Language Design Mode*: The language files of the graphical DSL can be created, modified and executed at runtime. Changes are interpreted and reflected directly in the behavior of the IME, enabling live metamodeling of the syntax and semantics. In language design mode, the IME detects changes to meta-DSL files by using a file watcher that tracks files within the language folder (see *4*). If files have been modified or created, the parsing and transformation step is restarted, the meta-specification is rebuilt (see *5*), and the changes are then propagated directly to the client.

- *Language User Mode*: Model files associated with the graphical DSLs can be created and modified. Access to the language folder is denied. The meta-specification and associated semantic files are only accessible by the IME's process, not the language user. In addition, the meta-specification is built once at startup and locked for the rest of the IME execution.

If the meta DSL files change, the meta specification must also be reconstructed by the server and propagated to the clients. This also means that the internal components in the client, such as the user interface, must be adapted to the changes. This affects not only the palette of model elements, buttons and context menu entries that are linked to the semantics of the model elements, but also the concrete syntax and visual representation of the model element types themselves on the canvas. Previously used model element types may be removed from the meta specification. Elements of these types remain in the model file, though they are no longer part of the language. These elements have no semantic or syntactic effect. They are visualized as red rectangles labeled with an identifier to mark them as outdated. Once a type is reintroduced, its associated elements can be used again. In addition to these GUI components, the behavior model of the IME itself must also reflect the current handling of constraints, as well as the specific file types associated with the model files, e.g. so that the Theia Editor opens a canvas instead of a text editor for a file type of a model. Since the GLSP handles some of these aspects rather statically, some components had to be re-implemented to dynamically adapt to the information of an evolving graphical DSL. The GLSP uses Sprotty[12] to render diagrams, which uses the JavaScript Syntax Extension (JSX)[13] to render elements on the modeling canvas. An interpreter has been implemented that maps the styles and appearances specified in the MSL to components of the JSX specification at runtime. This results in a variety of ways to dynamically visualize model elements, as shown in Figure 7.

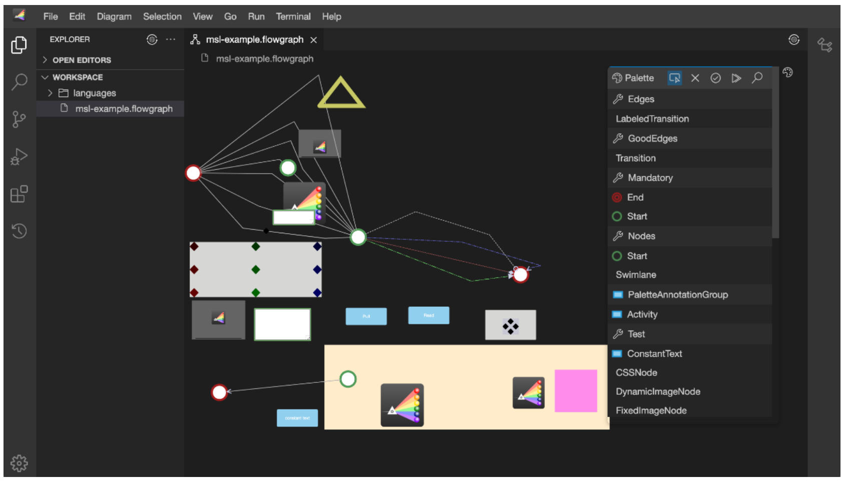


Figure 7: The model editor of an IME, showing a wide range of model element visualizations.

4.2 Interpreting Semantics at Runtime

To interpret the semantics of a DSL, JavaScript files can be integrated into the IME at runtime by using the eval function (see Figure 8), allowing code to be dynamically executed at runtime. At server startup, all JavaScript files in the language folder are read and integrated into the IME.

[12] https://sprotty.org

[13] https://facebook.github.io/jsx/

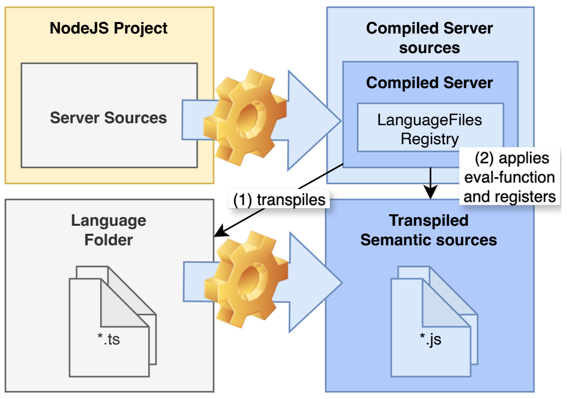


Figure 8: The implementation of interpreting semantics for an IME.

To provide the ability to develop TypeScript-based semantics in language design mode, changes to semantic files are tracked by a file watcher and transpiled into JavaScript by the TypeScript compiler on demand (see *1*). When a file changes, it is marked dirty by tracking its path in the file system, and after a semantic triggering event occurs, the dirty semantics are reintegrated by updating and replacing the obsolete semantics. Such semantics are reintegrated using the eval function and registered in a registry, called the *Language Files Registry* (see *2*). This registry keeps track of all the integrated semantic files. This reduces the number of computationally expensive eval calls [R+11] and serves as a central source of semantics for the IME components. The GLSP forms the foundation for working with graphical models by providing several handlers and actions. Its functionality includes context menus, the execution of create and delete operations and various user interactions such as double-clicking. The implementation of the general-purpose IME extends this functionality with some additional actions. These include an implementable generator, the preparation of attributes for a property view, the dynamic creation, labeling and categorization of the palette elements, as well as the support of custom file types and codecs for the associated model files. Currently, language engineers must develop the action handlers independently of the language specification of their designed graphical DSL. The handlers must be implemented to handle the shallow model element types provided by the meta-specification. *Recently an approach is being experimented with to dynamically generate and interpret the API classes from the meta-specification, but this is at an early stage.* At runtime, to link the implemented semantics of the graphical DSLs with the associated interaction and model elements, a handler manager is used within the IME accessing the *Language Files Registry*. Thus, when a user performs a double-click, the associated action is forwarded to the handler manager, which identifies the associated model element and semantics. Afterwards, the associated language semantics of the model element are executed.

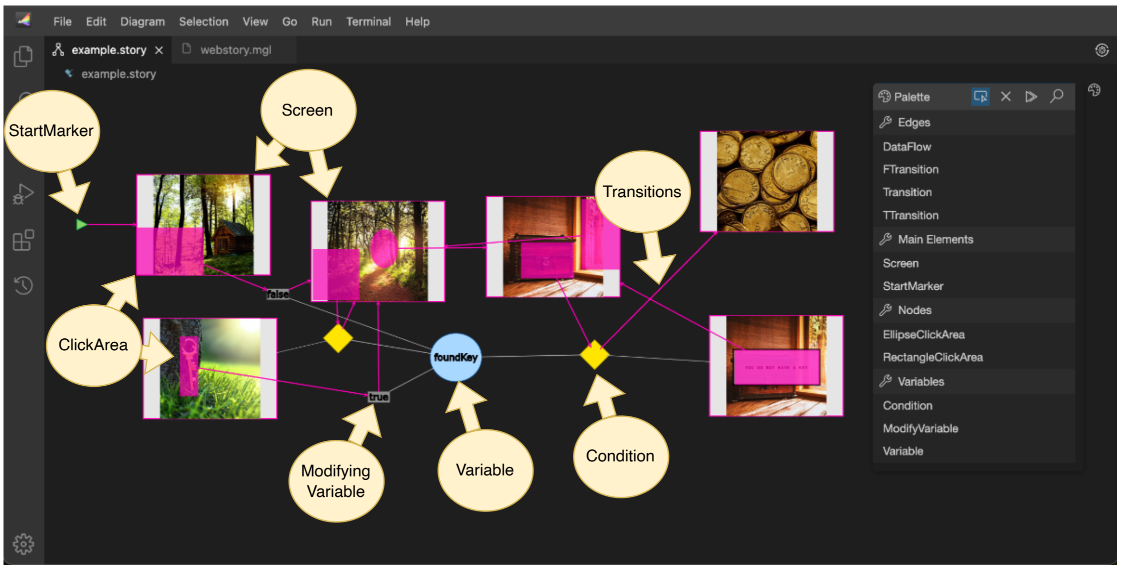


Figure 9: A model using the WebStory DSL and its model elements.

5 Evaluation in Practice

To evaluate the interpretive approach, this section compares it with the generative approach of Cinco Cloud[14]. Subsequently, the interpretive approach will be compared with the approaches of other existing tools. For this purpose, an exemplary graphical DSL was used, the WebStory [L+18], which has already been part of other case studies and examples [B+22, B+23]. For comparison, it was fully metamodeled, including the implementation of a generator, and a corresponding model was created. A virtual machine with an *Intel(R) Xeon(R) CPU E5-2640 v3 @2.60GHz and 8 GB RAM* was used for all experiments and to record the compile time of the generative approach.

5.1 Case Study: The WebStory

The WebStory [L+18] is a graphical DSL used to model web-based point-and-click adventures (see Figure 9), and essentially consists of nodes representing screens and click areas that can be placed within those screens. Transitions between screens are realized by directed edges from one click area to another screen. To model complex internal state transitions additional model elements are provided, i.e. variables, conditions, and modifying variables, as well as transitions for the associated data flow between these nodes and conditional transitions. If language engineers start metamodeling the WebStory, in both the generative and interpretive approaches, they first uses the meta-IME editor to design the language by creating meta-DSL files and specifying model element types, styles, and appearance using MGL and MSL. When it comes to evaluating the design and behavior of the language, the generative and interpretive approaches differ: The

[14] For the generative approach, the following project was used: https://gitlab.com/scce/cinco-cloud/-/tree/CincoCloudPyroArchived

generative approach requires the generation of a dedicated IME, resulting in an expensive overhead. The interpretive approach immediately reflects the changes in the general-purpose IME without leaving the project. In both cases, if the graphical DSL is usable, the context menu of the Theia IDE offers the option to create a WebStory model file. The interpretive approach automatically adds this option to the context menu when the file extension (e.g., *.story) is specified in the MGL. The user then opens the model file to access a modeling canvas, a palette of elements, and a properties view for editing. While the generative approach requires rebuilding the IME from scratch for each change, the interpretive approach allows seamless updates, with MGL and MSL changes directly adjusting the palette, constraint behavior, and visual representation of the elements in the editor. The interpretive approach allows for a responsive implementation of semantics for graphical DSLs, similar to syntax metamodeling. A WebStory generator is developed by annotating the graph model, implementing a TypeScript file, and placing it in the languages folder for automatic transpilation into JavaScript. As soon as the generator is valid and annotated in the MGL, a generator button appears in the user interface. Language engineers can immediately evaluate the behavior of the generator by running it on a model. Unlike the generative approach, which requires rebuilding and switching IMEs, the interpretive approach supports live development without rebuilding the entire IME. Evaluating the performance, the generative and interpretive approaches in Cinco Cloud make a significant difference in the development of graphical DSLs. The interpretive approach allows direct testing of a DSL without generating a new IME for each change, reducing compilation time and artifact size of the DSL. No performance losses in terms of usability were observed when modeling in the IME using the interpretive approach compared to the generative approach.

Since Cinco Cloud keeps a compressed artifacts of a language in an object storage for deployment and provisioning to domain experts, the size reduction of the artifact is important. While the generative approach produces a 71.8 MB artifact (65.4 MB compressed) after 4.5 minutes of compilation, the interpretive approach reduces the artifact size to 57 KB (10 KB compressed) with no compilation time. This reduces the language footprint by a magnitude of about 1289% (approx. 6696% compressed), while eliminating the recurring compilation time. This also implies a reduction of the IME including a language by a factor of about 2.6 (general-purpose IME of 27.4 MB, with a server size of 17.8 MB and client size of 9.8 MB). Thus, only one static IME compilation is required, optimizing efficiency and resource usage.

5.2 Related Approaches

In the following, the approaches of two existing tools are examined and compared with the approach presented in this paper. It is known that there are more tools in the field of MDE [E+13], but this evaluation is limited to more recent and web-based approaches. In particular, the approaches of the tools for metamodeling are considered with regard to the creation of abstract and concrete syntax as well as the development of semantics.

5.2.1 Eclipse Sirius Web

Eclipse Sirius Web[15] [G+24, B+21] is a cloud-based tool for creating and deploying IMEs, supporting not only graphical DSLs but also tables, forms, and other entities (see Figure 10). Users can graphically specify domains (similar to MGL) and views (extending beyond MSL)

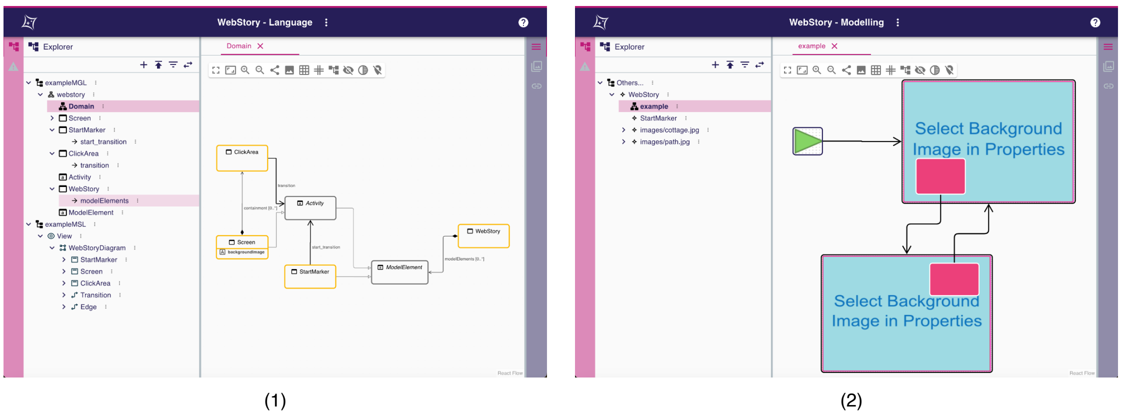

WebStory - Language
Explorer
Domain
WebStory - Modelling
Explorer
example
Select Background Image in Properties
Select Background Image in Properties
(1)
(2)

Figure 10: The development and usage of the *WebStory* DSL inside Eclipse Sirius Web.

using tree views, graphical editors, or forms (see *1*). Changes to the associated language update the persisted metamodel. To use or inspect a DSL (see *2*), users open an associated project, allowing immediate use of the language. While Sirius Web allows a language to be specified and used within the same project, it does not support live metamodeling, so projects must be closed and reopened to reflect changes[16]. The approach proposed in this paper improves on this by allowing immediate reflection of changes within the same project. Sirius Web supports semantics integration through compiled backend services or custom representations (e.g., textual editors[17]). However, these require separate compilation and deployment. In comparison, the proposed approach allows simultaneous live development of both DSL syntax and semantics without recompilation, offering a more seamless experience.

5.2.2 WebGME

WebGME[18] [MKK+14] integrates metamodeling and modeling in a single web application (see Figure 11). It enables live updates and seamless transitions between specifying and using a language. Metamodels, including nodes, relationships, and constraints, are graphically modeled, with changes instantly updating the persisted metamodel (see *1*). Specified nodes are accessible via a palette and can be used for modeling on a canvas or in a tree view (see *2*). WebGME also supports live metamodeling of concrete syntax through sidebar forms, allowing changes (e.g., color or shape) to apply globally across all associated models. Semantics are implemented as

[15] https://eclipse.dev/sirius/sirius-web.html
[16] http://docs.obeostudio.com/2024.3.0/help_center.html#limitations
[17] http://blog.obeosoft.com/langium-sirius-web
[18] https://webgme.org/

JavaScript plugins stored in a folder, similar to the proposed approach's language folder. Plugins are registered at runtime and executed via button clicks. However, plugin updates require refreshing the browser. In comparison, the proposed approach allows live development of semantics within the same project without browser refreshes, providing a smoother experience.

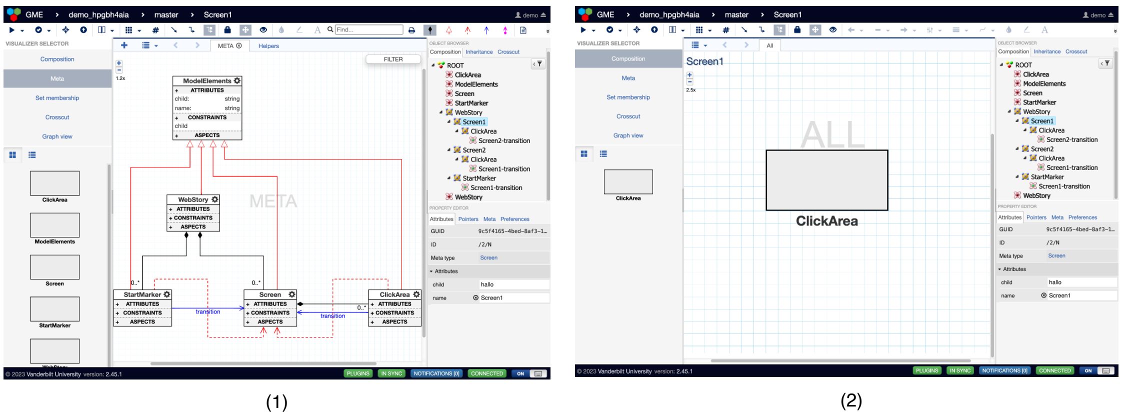


Figure 11: The development and usage of the *WebStory* DSL inside WebGME.

5.2.3 Results of the Evaluation

The comparisons of the approaches from the prior sections, results in the following table:

Evaluation of Functionality			
Approach	general-purpose IME	live metamodeling of syntax	live development of semantics
Cinco Cloud (generative approach)	No	No	No
Sirius Eclipse Sirius Web	Yes	No, Need to close and reopen project	No, need to be compiled into IME
WebGME	Yes	Yes	No, browser needs refresh
This Approach	Yes	Yes	Yes

Figure 12: The results of the comparison between the proposed approach, Cinco Cloud's generative Approach, Eclipse Sirius Web and WebGME.

6 Conclusion

This paper proposes a static, general-purpose IME that interprets metamodels of graphical DSLs. It replaces the generative approach of Cinco Cloud to address the expensive compilation times that occur whenever a language changes or an associated IME needs to be rebuilt. The approach introduces the ability of live metamodeling the syntax and semantics of graphical DSLs. This is done by unifying the involved language specifications (meta-DSL files) into a unified structure, referred to as a meta-specification, which is provided to the IME at runtime whenever the language changes. It is described how the meta-specification is constructed, including the eager resolution of all inherited properties of specified model element types to reduce computational cost. Besides the syntactic aspect, an approach for interpreting semantics is presented, allowing language engineers to develop semantics live within a meta-IME. This is done by integrating an existing GPL interpreter into the general-purpose IME. To illustrate the concepts in action, an implementation is given that introduces additional execution modes: the language design mode and the language user mode. The former allows the creation and modification of graphical DSLs and modification of the syntax and semantics. The latter restricts this possibility for a domain expert by interpreting the files only at startup and restricting their access at runtime. Evaluations against Cinco Cloud's generative approach [B+22] and tools such as Eclipse Sirius Web [G+24, B+21] and WebGME [MKK+14] showed significant improvements: elimination of 4.5 minutes of compilation time, reduction of the artifact size of a language from 72.2 MB (with 65.4 MB in compressed form) to 57 KB (with 10 KB in compressed form), an improvement by a magnitude of 1289% (6696% in compressed form), and support for live metamodeling without requiring projects to be reopened or refreshing the browser.

7 Future Work

The general-purpose IME presented in this paper has the potential to allow the ability to develop graphical DSLs even more responsive. Therefore, several potential improvements emerged from this approach that have not yet been addressed. As mentioned in Section 4, language engineers must currently use the shallow types of the meta-specification to implement semantics. We currently explore ways on how to make the development of semantics more domain-specific, e.g. by generating and interpreting API classes from the meta-specification at runtime. Beside that, previous work has demonstrated graphical DSLs for metamodeling [MKK+14]. These could be combined with custom codecs for the meta-DSL files, and live metamodeling capabilities, resulting in a form of projectional editing [V+14], where information from the meta-level is propagated directly into the IME, potentially resulting in *projectional live metamodeling*. Apart from that, Artificial Intelligence (AI) assisted programming realized by ChatGPT[19] and Github Copilot[20] moves traditional development towards natural language programming. With the now existing responsiveness of live metamodeling, new potentials arise to realize AI assisted (meta-)modeling or even natural language (meta-)modeling. It needs to be further investigated on how and which parts of the metamodeling process can be supported by the use of AI.

[19] https://chatgpt.com/
[20] https://github.com/features/copilot

Acknowledgements: This work was conducted with the financial support of Research Ireland Centre for Research Training in Artificial Intelligence under Grant No. 18/CRT/6223.

Bibliography

[B+84] A. D. Birrell et al. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2(1):39–59, 1984. https://doi.org/10.1145/2080.357392.

[B+16] S. Boßelmann et al. DIME: A Programming-Less Modeling Environment for Web Applications. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA*. Pp. 809–832. Springer International Publishing, Cham, 2016. http://dx.doi.org/10.1007/978-3-319-47169-3_60.

[B+21] F. Bedini et al. A generative Approach for creating Eclipse Sirius Editors for generic Systems. In *2021 IEEE International Systems Conference (SysCon)*. Pp. 1–8. 2021. https://doi.org/10.1109/SysCon48628.2021.9447062.

[B+22] A. Bainczyk et al. Cinco Cloud: A Holistic Approach for Web-Based Language-Driven Engineering. In *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering: 11th International Symposium, ISoLA*. Pp. 407—425. Springer-Verlag, Berlin, Heidelberg, 2022. https://doi.org/10.1007/978-3-031-19756-7_23.

[B+23] D. Busch et al. ChatGPT in the loop: a natural language extension for domain-specific modeling languages. In *International Conference on Bridging the Gap between AI and Reality*. Pp. 375–390. 2023. https://doi.org/10.1007/978-3-031-46002-9_24.

[E+13] S. Erdweg et al. The state of the art in language workbenches: Conclusions from the language workbench challenge. In *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings 6*. Pp. 197–217. 2013. https://doi.org/10.1007/978-3-319-02654-1_11.

[Fow10] M. Fowler. *Domain-specific languages*. Addison-Wesley Professional, September 2010. ISBN: 978-0-321-71294-3.

[G+24] T. Giraudet et al. Sirius Web: Insights in Language Workbenches-An Experience Report. *The Journal of Object Technology*, 2024. https://doi.org/10.5381/jot.2024.23.1.a6.

[Ken02] S. Kent. Model Driven Engineering. In Butler et al. (eds.), *Integrated Formal Methods*. Pp. 286–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. https://doi.org/10.1007/3-540-47884-1_16.

[L$^+$18] M. Lybecait et al. A Tutorial Introduction to Graphical Modeling and Metamodeling with CINCO. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018*. Pp. 519–538. Springer International Publishing, Cham, 2018. https://doi.org/10.1007/978-3-030-03418-4_31.

[MKK$^+$14] *Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure*. Volume 1237. Valencia, Spain, 09/2014 2014.

[N$^+$69] P. Naur et al. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels*. Scientific Affairs Division, NATO, 1969.

[N$^+$18] S. Naujokat et al. CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. *Int. J. Softw. Tools Technol. Transf.* 20(3):327–354, jun 2018. https://doi.org/10.1007/s10009-017-0453-6.

[R$^+$11] G. Richards et al. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In Mezini (ed.), *ECOOP 2011 – Object-Oriented Programming*. Pp. 52–78. Springer Berlin Heidelberg, 2011. https://doi.org/10.1007/978-3-642-22655-7_4.

[S$^+$24] B. Steffen et al. Language-Driven Engineering An Interdisciplinary Software Development Paradigm. *arXiv preprint arXiv:2402.10684*, 2024. https://doi.org/10.48550/arXiv.2402.10684.

[Sch06] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39(2):25–31, feb 2006. https://doi.org/10.1109/MC.2006.58.

[T$^+$21] T. Tegeler et al. An Introduction to Graphical Modeling of CI/CD Workflows with Rig. In *Leveraging Applications of Formal Methods, Verification and Validation: 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA*. Pp. 3–17. Springer International Publishing, 2021. https://doi.org/10.1007/978-3-030-89159-6_1.

[V$^+$14] M. Voelter et al. Towards user-friendly projectional editors. In Combemale et al. (eds.), *International Conference on Software Language Engineering*. Pp. 41–61. Springer International Publishing, 2014. https://doi.org/10.1007/978-3-319-11245-9_3.

[W$^+$23] A. Wasowski et al. *Domain-Specific Languages: Effective modeling, automation, and reuse*. Springer, 2023. https://doi.org/10.1007/978-3-031-23669-3.

[Z$^+$21] P. Zweihoff et al. Pyrus: An Online Modeling Environment for No-Code Data-Analytics Service Composition. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation: 10th International Symposium, ISoLA 2021*. Pp. 18–40. Springer, 2021. https://doi.org/10.1007/978-3-030-89159-6_2.