Virtual Domain Specific Languages via Embedded Projectional Editing

Niklas Korz
acm@korz.dev
Alugha
Germany

Artur Andrzejak
artur.andrzejak@uni-heidelberg.de
Heidelberg University
Germany

Abstract

Domain Specific Languages (DSLs) can be implemented as either *internal* DSL, i.e. essentially a library in a host general-purpose programming language (GPL), or as *external* DSL which is a stand-alone language unconstrained in its syntax. This choice implies an inherent trade-off between a limited syntactic and representational flexibility (internal DSLs), or an involved integration with GPLs and the need for a full stack of tools from a parser to a code generator (external DSLs).

We propose a solution which addresses this problem by representing a subset of a GPL - from simple code patterns to complex API calls - as GUI widgets in a hybrid editor. Our approach relies on matching parametrized patterns against the GPL program, and displaying the matched parts as dynamically rendered widgets. Such widgets can be interpreted as components of an external DSL. Since the source code is serialized as GPL text without annotations, there is no DSL outside the editor - hence the term 'virtual' DSL.

This solution has several advantages. The underlying GPL and the virtual DSL can be mixed in a compositional way, with zero cost of their integration. The project infrastructure does not need to be adapted. Furthermore, our approach works with mainstream GPLs like Python or JavaScript.

To lower the development effort of such virtual DSLs, we also propose an approach to generate patterns and the corresponding text-only GUI widgets from pairs of examples.

We evaluate our approach and its implementation on use cases from several domains. A live demo of the system can be accessed at https://puredit.korz.dev/ and the source code with examples at https://github.com/niklaskorz/puredit/.

CCS Concepts: • **Software and its engineering → Domain specific languages**; *Visual languages*; *Integrated and visual development environments*; *Programming by example.*

GPCE '23, October 22–23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0406-2/23/10.
https://doi.org/10.1145/3624007.3624059

Keywords: DSLs, Projectional editing, Programming language integration, Programing by example, Assisted editing and IntelliSense

ACM Reference Format:
Niklas Korz and Artur Andrzejak. 2023. Virtual Domain Specific Languages via Embedded Projectional Editing. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3624007.3624059

1 Introduction

Domain Specific Languages (DSLs) have proven useful in developing software systems, and are increasingly adopted by practitioners in a multitude of application domains [16, 25, 43]. They can clearly communicate the intent of a part of a system, hide irrelevant implementation details, and make it harder to say "wrong things" in code. These properties help to improve development productivity and prevent defects. Finally, DSLs can effectively facilitate communication between domain experts and developers.

DSLs can be implemented as either *internal* DSL, i.e. essentially a library in a host general-purpose programming language (GPL), or as *external* DSL which is a stand-alone language unconstrained in its syntax. Internal DSLs can be implemented more easily yet and naturally integrate in the host GPL. However, they offer only a limited syntactic and representational flexibility which is tightly coupled with the syntax of their host language. On the other hand, while not limited in their syntax, the external DSLs face an issue of complex and possibly inefficient interfacing with other languages in the project. Moreover, they feature a high cost of implementation due to a need for a full stack of tools from a parser to code generator or compiler.

Projectional editors [56, 66, 67] solve the problem of integrating different languages by being able to render each language independently yet side-by-side. They can be also enhanced with interactive, graphical elements such as tables or dynamic diagrams that suit the specific needs of a certain domain. However, projectional editors come with an increased development and maintenance effort for the DSL developers, and have usability issues [67], such as challenges of efficient entering textual code, or code modifications. In

```
((table) => {
  console.log("Replacing ...");
  table.column("name").replace("Mister", "Mr.");
})
(db["students"]);
```

Listing 1. An example code in TypeScript.

```
change table students
  console.log("Replacing ...");
  table.column("name").replace("Mister", "Mr.");
end change
```

Listing 2. Hybrid representation of code from Listing 1. Projections are shown in bold.

addition, code is typically serialized in a proprietary format, making infrastructure integration difficult.

To address these issues, we propose an approach which extends a textual code editor with the ability to represent parts of code as embedded textual or graphical GUIs. Listing 1 and Listing 2 illustrate this solution. Listing 1 shows a fragment of TypeScript code as rendered in a conventional editor or IDE. Lines 1 to 5 define an anonymous function with a parameter `table` and function body in lines 2 and 3. Line 5 applies this function to an object `db["students"]` representing a database table. Listing 2 displays how the same program fragment is shown in our *hybrid editor*. Lines 1 and 4 in the latter listing are part of a *projection*, an embedded GUI which offers a developer-friendly representation of the source code lines 1, 4, and 5 from Listing 1. Projections can be interpreted as components of an external DSL. Their representation can assume any form - from text to tables to diagrams to equations.

Within the projection shown in Listing 2 users can change the argument `students` by typing it or selecting from a list of recommendations (not shown). They can also edit the textual code representation in lines 2 and 3 as usual. Users can enter new (textual) projections in their code by using the code recommendation feature of the hybrid editor. For example, after typing 'c' in an empty line in the live demo the projection 'change table' is proposed. It is also possible to edit the hybrid and the traditional form of GPL source code side by side, with updates being immediately synchronized. Grammar errors in the GPL code lead to failure of identifying the correct projection. As a consequence, the hybrid editor shows the original (erroneous) GPL code instead.

We illustrate the flexibility of representation in the hybrid editor with a more complex example - visual editing of mathematical expressions. We created a small Python library `mathdsl` (a wrapper of few large libraries) to support conversion of formulas expressed in LaTeX into executable NumPy code (see Section 5.2). Listing 3 shows code which implements a rotation of a 2-dimensional vector by angle

```
import mathdsl
rotate, args = mathdsl.compile("\\begin{pmatrix}\\cos\\
    theta & -\\sin\\theta\\\\ \\sin\\theta & \\cos\\theta
    \\end{pmatrix}\\begin{pmatrix}x\\\\ y\\\\end{pmatrix}")
print("rotate(x, y, theta):")
print(rotate(x=1, y=2, theta=0.5))
```

Listing 3. Rotation of a two-dimensional vector using library `mathdsl`.

```
1  import mathdsl
2  rotate, args = ⎛cos θ  −sin θ⎞⎛x⎞
                  ⎝sin θ   cos θ⎠⎝y⎠
3  print("rotate(x, y, theta):")
4  print(rotate(x=1, y=2, theta=0.5))
```

Figure 1. Code from Listing 3 as shown in the hybrid editor.

θ using `mathdsl`. While this representation relieves users familiar with LaTeX from coding in NumPy, the code in line 2 it is not easy to write and understand.

Figure 1 shows a representation of the same code in our hybrid editor. Users can enter each part of mathematical expression as a LaTeX code and/or via a visual palette/keyboard (opened by an icon at the end of line 2). Both code comprehension and the ease of editing are enhanced. A reader is encouraged to try out this example in the live demo. Our prototype also supports integration of such a solution in JupyterLab.

An essential feature of our approach is that the source code retains its traditional textual form while persisted, without any annotations or changes (internally, code is represented as an equivalent AST). In this way, the hybrid editor achieves full compatibility with existing infrastructure tools like linters, compilers, or other editors. Furthermore, developers do not need to adapt their code to use the system. Effectively, there are no traces of a DSL outside the editor - hence the term 'virtual' DSL.

Under the hood, the hybrid editor tries to find in the source code fragments which match one of the predefined *patterns*. In Listing 1, lines 1 and 5 match such a pattern. It is defined together with the corresponding *projection template*, here the one responsible for displaying lines 1 and 4 in Listing 2. Our hybrid editor renders a projection for each matching pattern (with corresponding parameters, e.g. `students`). As outlined in Section 2.3, edits of the textual or projectional representation update the internal AST model and might trigger a redraw of the view.

A pattern and a projection template constitute together a *DSL component* which is responsible for an alternative representation of a specific aspect of the GPL language or its library. Such components can be developed separately, allowing to incrementally grow a virtual DSL in a modularized

way. Compared to an external DSL, the development effort for a virtual DSL is a typically smaller.

To lower the development cost even further we propose an approach to generate patterns and the corresponding text-only projection templates from multiple examples. A DSL developer provides pairs of a GPL code to be matched and desired textual projection. Our algorithm either requests for more/other examples to resolve ambiguity or inconsistencies or generates a DSL component (i.e. pattern and corresponding projection template). These artifacts can be either used directly or can serve as a basis for further refinements.

In addition to work on projectional editing [9, 15, 37, 49, 52, 65, 67] there is already a substantial body of research on enriching *textual* editors with interactive GUI-based components for developer-friendly representation of code [1, 19, 54, 55, 59]. As discussed in Section 6, most approaches require adaptation of the source code, and only few target mainstream programming languages (and if so, predominantly Java). Our work focuses on practical usability of such a solution, in particular support for mainstream GPL languages like Python or JavaScript, compatibility with an existing infrastructure, and a moderate development effort of virtual DSLs. To further increase the practical value, our prototype uses web-based technologies and is implemented in TypeScript. It works almost out-of-the-box in JupyterLab and can be easily adapted for Visual Studio Code or other web-based editors.

The main contributions of this work are:

- Approach for specification of patterns for matching code fragments in GPL source code.
- Algorithm for matching code fragments and extraction of arguments from code.
- Approach for efficient rendering and editing of the projections associated with patterns, and bidirectional updating of textual and projectional code representation.
- Prototypical implementation of a web-based hybrid editor as a stand-alone tool and a JupyterLab extension.
- Approach and an implementation for synthesizing DSL components (with text-only projections) from samples of pairs GPL code/textual projection.
- An evaluation via a proof-of-concept for the domains spreadsheet processing and formula editing in GPL languages TypeScript and Python.

This paper has the following structure. Section 2 details the virtual DSL approach. Section 3 describes the algorithm for generating DSL components from examples. Section 4 outlines the implementation. Section 5 describes the evaluation. Section 6 discusses related work and Section 7 contains conclusions.

```
1 let db = contextVariable("db");
2 let tbl = arg("table", "string");
3 let [changePattern,changeDraft] =
4   statementPattern`
5   ((table) =>
6     ${ block({tbl:"table"}) })
7   (${ db }}[${ tbl }]);`
8 ;
```

Listing 4. A description in the templating language of a pattern which matches parts of code in Listing 1.

2 Virtual DSL - Concepts and Algorithms

A virtual DSL is a system consisting of a hybrid editor (same for all projects), and a set of DSL components to be developed for a specific scenario. Recall from Section 1 that each DSL component is a pair including a pattern (defined rigorously in Section 2.1) and a template projection. A pattern matches in the edited program a code fragment called a *projected code (fragment)*. Such fragments can be simple expressions or statements, calls to APIs or libraries, or even statements in an internal DSLs. The projection templates are essentially GUI components (widgets), in our implementation using the Svelte web framework.

Section 2.1 describes how patterns are defined. In Section 2.2 we outline the process of compositional matching of these pattern and the extraction of arguments for projections. Section 2.3 describes the mechanisms related to rendering and editing projections.

2.1 Patterns and Templating Language

A *pattern* is a pair of an AST in a target GPL and an optional set of data structures called *active nodes*. An active node consists of a regular AST-node with a unique identifier and a separately maintained data structure referenced by this identifier. Active nodes play a special role during the matching process explained in Section 2.2 as they capture specific sets of AST nodes or provide context information.

Since it is difficult to create and edit a pattern directly, we use a subset of TypeScript called *templating language TL* for generating and representing patterns. A pattern description in TL is just a fragment of code in TypeScript which contains one *tagged template literal* [32, sections 13.2.9 and 13.3.11] and optionally declares some objects using functions `arg(argName, nodeType)`, `block(contextVars)`, or `contextVariable(varName)`. The role of these objects is to create active nodes in the described pattern.

We illustrate the above concepts on an example. Listing 4 shows a TL description of a pattern matching lines 1, 4, and 5 in Listing 1, i.e. code shown as a projection in lines 1 and 4 of Listing 2. Lines 4 to 7 in Listing 4 contain the tagged template literal which consists of a call to our custom function `statementPattern` and a template literal, i.e. essentially a string with parameters to be interpolated (parameters are

shown highlighted). In TypeScript, these parameters are any expressions enclosed in `${..}`.

The template literal represents the AST to be matched. All string parts (i.e. parts outside `${..}`) will be turned to AST nodes matched literally, and each parameter will give rise to an active node. Consequently, a pattern created by Listing 4 will have three active nodes. The latter are created by the calls: `block({tbl:"table"})` (line 6), `contextVariable("db")` (lines 1 and 7), and `arg("table", "string")` (lines 2 and 7), in this order. The details of these functions are described in Section 2.1.1. Note that the string parts of the template literal express the code of a target GPL language, and could be e.g. Python, not necessarily TypeScript as in this example.

In essence, the first active node captures an inner block of target GPL statements enclosed by a matched source code fragment. In in Listing 1 this block is in lines 2-3. The second active node, `db`, is a context variable potentially intialized in a surrounding pattern match. It specifies a name of a variable giving access to a database specified in the surrounding code. This access can be used by the hybrid editor to e.g. dynamically provide a list of valid table names for parameter recommendations within the current projection. Finally, the third active node, `tbl`, has the task of capturing and updating a value of a source code AST node of type `string`. It is then used as a parameter to be shown and edited in the corresponding projection. In this case, the value is "students" (line 5 in Listing 1), and the same value is shown in the projection in Listing 2 (line 1).

A TL description like above is converted by our function `statementPattern` to a pattern. In this process, each parameter of a tagged literal template generates an active node. The corresponding internal data structure is kept for the matching process (Section 2.2), and the parameter location (substring `${..}`) is replaced by a unique identifier referencing this data structure. The literal template is then turned into a normal string containing a fragment of the target GPL code. We parse it subsequently into an AST using the parser generator tree-sitter [62] into TypeScript, Python, or (in future) other target GPL.

2.1.1 Details of Active Nodes.

We describe in the following the three kinds of active nodes of a pattern and the corresponding functions used in the templating language TL to generate them.

Argument active node. This kind of node is specified in TL via the helper function `arg(argumentName, nodeType)`. Its purpose is capturing the text content of an identifier or literal nodes in the GPL source code as an argument for the corresponding projection. Moreover, depending on the value of `nodeType`, we can also capture arbitrary subtrees by specifying a filter function. The type specification `nodeType` allows to match AST-nodes by type and not their text content. Possible node types depend on the target GPL and are specified by tree-sitter. Furthermore, the captured value is accessible to context variables under the `argumentName`. The parameter `argumentName` sets the name under which a projection can access the value captured in the source code.

Block active node. This kind of node is created in TL by the helper function `block(contextVariables)`. It captures nested blocks of code, such as the body of a function definition or the branch of an if-clause. Additionally, blocks themselves are recursively searched for pattern matches, in order to support compositionality. To this end optional context variables (explained below) can be passed down to patterns matching inner code.

Context variable active node. In TL, such nodes are expressed via the function `contextVariable(variableName)`. They declare context variables used in the pattern matches within inner code blocks. For example, the name of the currently specified table (in outer code block) can be passed via such variables to patterns matching code within inner code blocks. This can be helpful for e.g. recommendations of column names in the inner projections. When a pattern is initialized for the matching process, each such active node is replaced by a value specified by the corresponding context variable. The latter are set in a surrounding context by argument active nodes. Furthermore, context variables can also be declared and initialized globally for the whole editor, which is useful for providing the names of global variables to the projections.

2.2 Pattern Matching on Abstract Syntax Trees

Contrary to pure projectional editors which store programs in form of serialized trees (e.g. XML or JSON files), the hybrid editor serializes the source code in conventional text-based form, internally maintaining an equivalent AST. Rendering requires dynamic detection of projected code fragments matched by the patterns, extraction of the relevant information, and dynamic creation or update of each projection.

Algorithms 1 and 2 show the pseudocode of the approach for matching patterns in GPL code. We use the following symbols. *Src* is a syntax tree of the currently edited file and *Pat* is the set of the defined patterns. *Ctx* is a mapping from context variables to their values, initially populated with globally defined context variables.

Algorithm 1 visits each node n of the AST *Src* in a depth-first search fashion (line 3). For each n and each defined pattern p it checks via MatchPattern whether the subtree of *Src* starting at n matches the pattern p. The pseudocode does not show a small optimization such that for n only patterns are considered with the same type of root node as the type of n (see [42] for all details).

If there is match of p at n, the result set *Res* is updated (line 8). Furthermore, for each inner code block identified by this match we issue a recursive call of FindPatterns at

Algorithm 1 Top-level algorithm for matching patterns.

```
Require: Src, Pat, Ctx                          ▷ See text for definitions.
1:  function FindPatterns(Src, Ctx)
2:      Res ← ∅                                 ▷ Resulting set of matches
3:      for all n ∈ DFS(Src) do
4:          for all p ∈ Pat do
5:              Args ← ∅                        ▷ Set of captured arguments
6:              Blks ← ∅                        ▷ Set of captured blocks
7:              if MatchPattern(p, n, Args, Blks, Ctx)
                then
8:                  Res ← Res ∪ {(p, n, Args, Blks)}
9:                  ▷ Recursively search all inner code blocks ◁
10:                 for all (n̂, Ĉ) ∈ Blks do
11:                     Ŝ ← GetSubtreeAtNode(n̂, Src)
12:                     Res ← Res ∪ FindPatterns(Ŝ, Ctx ∪
                        Ĉ)
13:                 end for
14:                 break                       ▷ At most one match per n
15:             end if
16:         end for
17:     end for
18:     return Res
19: end function
```

a subtree $\hat{S}$, i.e. the root node of the code block, to support compositionality (lines 11-12).

The algorithm returns a set of *matches*. Each is a tuple $(p, n, Args, Blks)$ of a pattern p, root node n of an AST matching p, and sets $Args$, $Blks$ of captured arguments and captured blocks, respectively.

The pattern candidates are evaluated by function MatchPattern. In essence, a node n of Src matches a pattern when it is deeply equivalent to the pattern's root node, see the recursive descend at lines 24-29. This function also updates the sets $Args$ and $Blks$ according to the meaning of the active nodes (lines 2-9), see Section 2.1, and checks that the context variables have correct values in the candidate AST subtree (lines 10-13).

Algorithm 2 A function for testing whether an AST tree matches a pattern and extracting arguments and code blocks.

```
1:  function MatchPattern(p, n, Args, Blks, Ctx)
2:      if isArgumentNode(p) ∧ nodeType(n) =
        requiredType(p) then
3:          Args ← Args ∪ {(argumentName(p), n)}
4:          return true
5:      end if
6:      if isBlockNode(p) then
7:          Blk ← Blk ∪ {(n, blockCxtSet(p))}
8:          return true
9:      end if
10:     if isContextVariable(p) then
11:         v ← contextVariableName(p)
12:         return isIdentifier(n) ∧ v ∈ dom(Ctx) ∧
            Ctx(v) = text(n)
13:     end if
14:     if nodeType(p) ≠ nodeType(n) then
15:         return false
16:     end if
17:     ▷ Atomi (leaf) nodes must be compared by their texts ◁
18:     if isAtomic(p) then
19:         return text(p) = text(n)
20:     end if
21:     if |children(p)| ≠ |children(n)| then
22:         return false
23:     end if
24:     ▷ Children of p and n are assumed to be aligned ◁
25:     for all p̂ ∈ children(p), n̂ ∈ children(n) do
26:         if ¬ MatchPattern(p̂, n̂, Args, Blks, Ctx) then
27:             return false
28:         end if
29:     end for
30:     return true
31: end function
```

2.3 Rendering and Editing Projections

Given a set of matches $m_1, \ldots, m_k$ (output of Algorithm 1) and the predefined projection templates the hybrid editor is capable of rendering the edited program S. Essentially, all statements (or AST subtrees) in S not included in any of the matchings are identified as non-projected code. These parts of S are shown in a traditional textual form. On the other hand, the rendering of each projected code fragment is delegated to the respective projection.

The process of updating internal model of the hybrid editor requires more detailed explanation. When the source code changes due to edits of the textual or the projectional parts the internal model (i.e. source code state) of the hybrid editor becomes invalid. Incremental, i.e. local updates of this model are difficult to achieve due to presence of context variables. Therefore we need to rerun Algorithm 1 in order to re-match the entire AST against the predefined patterns, and re-render the hybrid representation.

To optimize this process, we reuse the instantiated projections created by the previous model if possible. To this end we need to compare source code ranges $r'_1, \ldots, r'_k$ of the projected code (obtained from the matchings $m'_1, \ldots, m'_k$) from the previous model against the projected code ranges $r_1, \ldots, r_k$ of the new model. If some r'_j and r_i overlap for the same pattern, we can reuse the previously instantiated projection after updating it with new argument values. In general, this provides a smooth user experience even for larger source code files.

```
1 tbl["name"] = tbl["name"].replace("Mister", "Mr.");
2 replace Mister with Mr. in column name
3
4 tbl["title"] = tbl["title"].replace("PhD", "Dr.");
5 replace PhD with Dr. in column title
6
7 tbl["adr"] = tbl["adr"].replace("Road", "Rd.");
8 replace Road with Rd. in column adr
```

Listing 5. Samples of projected code and the corresponding (textual) projection.

```
1 // Pattern prototype (AST unparsed)
2 tbl[<cp1>] = tbl[<cp2>].replace(<cp3>, <cp4>);
3
4 // Projection template prototype (list joined)
5 replace <pp1> with <pp2> in column <pp3>
```

Listing 6. Pattern and projection template prototypes derived from Listing 5. <> indicate placeholders.

3 Generating DSL Components from Examples

In this section we propose an algorithm for synthesizing simple patterns and projections from multiple samples of GPL code and textual projections.

While the development effort of a pattern and a corresponding projection template is modest (below 100 LOC of a typical case, see Section 5), a developer of a virtual DSL might be faced with a steep learning curve due to multiple frameworks/technologies. In our implementation, developing a pattern requires knowledge of TypeScript and Javascript's tagged template literals (see Section 2.1), and creating a projection template assumes familiarity with Svelte and optionally with HTML/CSS.

To this end we propose an approach to synthesize a basic form of a pattern and a corresponding projection template from multiple examples of pairs (projected code, projection). We assume that the projections are of textual form, like in lines 1 and 4 in Listing 2. Consequently, projection templates can be understood in context of this section as string templates where static text is interweaved with parts displaying variable and editable content called *placeholders*.

Listing 5 gives an example of 3 pairs of samples, each pair consisting of projected code and the corresponding expected textual projection. The code samples must be selected or created by the DSL-developer such that they differ in tokens indicating placeholders. Here the argument to `tbl["name"]` and both arguments to `replace()` indicate the differences. Analogously, differences between projection texts indicate positions of the placeholders. For the first pair, these positions are at the tokens `Mister`, `Mr.`, and `name`. By using the same placeholder values in a code sample and in it's projection we indicate how the placeholders should be mapped.

Given this input, our algorithm is able to generate a *derived pattern* and corresponding *derived (projection) template*. Latter is the core part of a GUI widget (in our prototype, a Svelte component) rendering the projection in the hybrid editor. Despite of some limitations, this solution covers many simple cases without a need for any coding. The generated code can be useful even for more experienced developers as a skeleton for further extensions.

The relevant limitations of the approach are (in addition to text-only form of projections) lack of context variables (Section 2.1.1). Moreover, a derived pair pattern/template can have at most one nested code block. A DSL-developer can still provide these features, they are not generally precluded. We opted for this simple form to reduce ambiguity and keep the number of required examples low.

In the following we detail our approach. Section 3.1 discusses how we identify constant and variable parts of GPL code by comparing the AST trees of the samples. In Section 3.2 we explain how projection samples are used to generate a textual template. Section 3.3 outlines mapping of placeholders found in code to those found in projections. Finally, Section 3.4 explains how this information is used to generate the pair pattern/projection template.

3.1 Analyzing Samples of Projected Code

The goal of analyzing samples is to obtain a *pattern prototype* defined as an AST-tree with some nodes marked as placeholders. Listing 6 (top) shows the unparsed, i.e. serialized pattern prototype derived from the code samples in Listing 5. Additionally, we compute a list of paths to the found placeholders in this prototype. This is required for mapping pattern and projection placeholders.

The algorithm works iteratively. Each sample code is parsed to an AST (which are formally also pattern prototypes). The comparison of two first ASTs yields a first candidate for pattern prototype, which is then compared against third code sample AST, yielding second candidate result etc. When all nodes of X are processed, we return the last candidate as the pattern prototype, and a list of paths to the placeholders.

A recursive comparison of two ASTs X and Y (or technically, two pattern prototypes) is the core of this routine. Given a list of AST nodes $x_1, \ldots, x_k$ (e.g. children of a node) from X and the same-length list $y_1, \ldots, y_k$ of nodes from Y, we test which node pairs (x_i, y_i) are identical and which differ. A pair is considered identical if both nodes have same AST-type, same content (e.g. identifier text), and are leaves. If x_i are y_i are same-type non-leaves with equal number of children, we perform a recursive descend on the set of their children.

Otherwise, x_i and y_i are considered different. They are marked as placeholders, and their tree path is recorded. If both have the same AST-node type, we inherit this type to

their placeholders. In other case it is considered a 'wildcard', i.e. the placeholder can match any type.

3.2 Analyzing Samples of Projections

Analogously to code samples, strings representing textual projections are compared for common (equal) and diverging parts. The goal is to obtain a *(projection) template prototype* which is an alternating list $c_0, v_0, c_1, v_1, \ldots$ of static strings c_i (for common parts) and placeholders v_i (for diverging parts). We assume here that two subsequent variable parts are always separated by a static part, e.g. a comma. Listing 6 (bottom) shows the template prototype derived from the projection samples in Listing 5 (list joined to one string).

First, each sample projection text is turned into a list of tokens. Note that this list can be considered a special case of a prototype. We set the first sample as the initial solution and then iteratively refine the current version p of a template prototype by comparing it against the next sample s.

At each iteration step, we use the Meyer's Diff algorithm [51] for detecting the longest common token subsequences between p and s. We exploit the fact that each longest common token subsequence c_i must be followed by a diverging part v_i (v_i might be optional for the last common subsequence). For each c_i found by Meyer's Diff algorithm we can thus add to the updated version p' of the prototype the string c_i as the subsequent static part, and the v_i as the next placeholder.

3.3 Mapping Placeholders from Code to Projections

The next goal is to map placeholders found in code samples (Section 3.1) to those found in projection samples (Section 3.2). To make the presentation clearer, we call the earlier *c-placeholders* cp and the latter *p-placeholders* pp. In Listing 6 we have c-placeholders $cp_1, \ldots, cp_4$ and p-placeholders $pp_1, \ldots, pp_3$.

Assume that we are given c-placeholders $cp_1, \ldots, cp_m$, p-placeholders $pp_1, \ldots, pp_n$, a projection template prototype t (found as in Section 3.2), a list V of tree paths to the c-placeholders (found as in Section 3.1), and the pairs of samples $(c_1, p_1), \ldots, (c_k, p_k)$. We want to find a relation M which maps each cp_i to exactly one pp_j, and each pp_j appears in the image of M (in other words, M is surjective). It means that multiple placeholders in the code can be 'connected' by the same value, but this group corresponds to only one placeholder in the projection.

We propose the following algorithm to this end. We iterate over the pairs of samples and build M incrementally. For (c_x, p_x) we compute a list Q of strings of p_x which correspond to the placeholders in the template prototype t (the list is ordered such that ith element corresponds to pp_i). We can achieve this by using again the Meyer's Diff algorithm on p_x and t: each divergent sequence of tokens corresponds to one p-placeholder, since common subsequences are identical static parts of p_x and t. For example, p_1 in Listing 5 gives Q = ["Mister", "Mr.", "name"] corresponding to pp_1, pp_2, pp_3 in t.

The strings in Q create a link between p-placeholders and c-placeholders since they should also appear in the code sample c_x. To exploit this, we iterate over V (the tree paths to c-placeholders), and for each $v_i \in V$ we retrieve the AST node or AST-subtree in c_x referenced by v_i. The text obtained from this AST fragment is searched in Q. If found at position j in Q, we have obtained a mapping of a c-placeholder cp_i to p-placeholder pp_j: $M(cp_i) = pp_j$. For example, after processing the first pair (c_1, p_1), we obtain the mapping M defined by $cp_1 \to pp_3$, $cp_2 \to pp_3$, $cp_3 \to pp_1$, $cp_4 \to pp_2$.

If different pairs of samples create incoherent mappings (i.e. a c-placeholder is mapped to different p-placeholders), the algorithm terminates with an error. We also check that all p-placeholders are covered. Finally, M is returned after all pairs of samples are processed.

3.4 Constructing a Pattern and a Projection Template

After the three processing steps outlined above a derived pattern and a corresponding projection template can be constructed. We save both artifacts a regular source code files which can be used or edited as any manually developed pattern/template pair.

A derived pattern is obtained from any code sample c_i (formally, a string) by replacing the substrings corresponding to c-placeholders by strings representing calls to helper functions discussed in Section 2.1.1. To this end we iterate over the AST paths identifying c-placeholders (found in Section 3.1), for each one we retrieve the substring range in c_i corresponding to this AST path, and replace it by the function call. Information about the source code range of each AST subtree is provided by default by tree-sitter.

A derived projection template is created from a skeleton of a GUI widget which displays text only and admits at most one nested block. The parametrized part is a list of tokens, where each token is either an HTML string for the static parts of the projection or a p-placeholder. Each p-placeholder is displayed as a `TextInput` component and 'wired' to the correct c-placeholder(s) (i.e. helper function calls) in the derived pattern according to the mapping M (Section 3.3).

4 Implementation Details

4.1 Overall Architecture

We use CodeMirror [8] web-based editor as a basis for our hybrid editor. In this environment, any components representable in HTML and JavaScript (in particular, any Svelte components) can be used as projection widgets. However, conceptually other extensible editors or IDEs such as Visual Studio Code or its basis library Monaco Editor [14] are a suitable basis for the hybrid editor.

Following this choice, the primary programming language of the project is TypeScript, with some complementary libraries like tree-sitter using other languages at their core. In particular, all algorithms described in Section 2 are implemented in TypeScript.

We make the hybrid editor available in two environments. The first is as a stand-alone browser-based editor with a web server as a backend (see the live demo). The other environment is JupyterLab [40], a popular tool data scientists. Since JupyterLab uses CodeMirror as its cell (code) editor, we can integrate our hybrid editor as a frontend extension, without affecting the Jupyter backend. This environment is used in both virtual DSLs discussed in Section 5.

4.2 Projections: Implementation and Editor Integration

Projections in the hybrid editor could be implemented by any of the web frond end frameworks based on JavaScript, like React, Angular, Vue, or Svelte. Using such generic frameworks for projections offer flexibility, thus potentially allowing even more special projections such as diagrams or editors for graphical user interfaces. We choose Svelte [11] to its reactivity, small size of transferred files, ease of debugging, and lifecycle events.

A key feature needed in a hybrid editor is the ability to replace ranges of text with custom display and editing components. CodeMirror offers this ability via *replacement decorations* [33] which can render any HTML content instead of a specified text range. As CodeMirror renders its contents using conventional HTML and CSS, it is possible to turn these elements into interactive widgets (i.e. projections) using JavaScript.

When the document undergoes updates, previous decorations are retained if they remain relevant in the updated positions. Consequently, the Svelte properties update to the new match, leading to an automatic refresh of the widget's contents. This approach ensures continuity and alignment with user input.

Our system understands that CodeMirror expects widget classes to adhere to a specific element-producing interface. To streamline this process, we've designed a wrapper function that automatically morphs Svelte components into the desired widget classes. This wrapper plays a crucial role, ensuring that all critical data, such as the pattern match, context, and editor state, is transferred and updated as Svelte properties whenever changes occur.

4.2.1 Input Management. User input management is pivotal in our system. A shared `TextInput` component handles user input directly, replacing the source code at the corresponding AST node. This action triggers a syntax tree rematch, causing the system to update the components with fresh match data. The underlying principle is maintaining the textual document as the authentic source for projections visible to the user, ensuring consistency. Replacing code with user input is a straightforward process, especially for string literals, where the requirement is merely replacing any quotation marks with their escaped variants. For more complex node types, we've implemented serialization functions that guarantee the produced code aligns with the field's pattern node.

To ensure the projections blend well with the rest of the document, cursor movement is monitored to provide a fluent transition between projection and code. We include a *focus manager* in every projection instance that is responsible for keeping track of the focusable inputs inside a projection widget and of the input that is currently focused. On internal cursor movement, the focus manager ensures the next or previous input element is focused, depending on whether the cursor has "left" the field on the start or end side. For the first and last input inside the projection, the cursor is moved to a position in the text editor adjacent to the projection instead. Overall, this improves the user experience by allowing modifications of the whole document without having to use a mouse.

New projections are inserted into the edited program through our hybrid editor's completion mechanism. As the user types a text sequence resembling a projection's name, they can choose a suggested projection to insert into the document. The system then generates a code snippet from the original code pattern template, replaces template placeholders with empty values, and integrates it into the document.

4.3 Simplified Definitions of Projections

The analysis of the development effort via counting non-comment Lines of Code (LOC) in Section 5 reveals that the majority of line count can be attributed to the code of a Svelte component. However, a closer look at the Svelte components reveals that these consist mostly of import statements and HTML code instead of logic. Consequently, we have introduced a wrapper function `simpleProjection`. It creates projectional widgets without need to understand Svelte components or the internals of the CodeMirror editor.

Listing 7, lines 12-15 shows an application of this function. It receives an array of tokens, where a token is either a string, a reference to a code placeholder *cp* (as defined in Section 3), or an array of such references if multiple source code parts are to be modified by a projection field. The placeholder reference must be named identically as the special element produced by the `arg` helper function for *cp* (Section 2.1) created when defining the associated pattern. In this way, we 'wire' the code placeholders with the projection placeholders and avoid typos.

Listing 7 shows a pattern definition in the TL (top) and a simplified projection definition (bottom) of a component for spreadsheet analysis DSL. The object references `fName`, `sName` specifying code placeholders (or special elements of type argument node) in the pattern definition are used in

```
// Pattern definition
const dsl = contextVariable("dsl");
const fName = arg("fName", "string");
const sName = arg("sName", "string");

export const [pattern, draft] =
pythonParser.statementPattern`
with ${dsl}.load_sheet(${fName}, ${sName}) as sheet:
  ${block({ sheet: "sheet" })}
`;

// Simplified definition of a projection
export const widget = simpleProjection(
  ["load sheet", sName, "from", fName, ":"]
);
```

Listing 7. Definitions of a pattern via Templating Language and its projection in a simplified form.

```
with dsl.load_sheet("Chromatography.xlsx", "raw data")
 as sheet:
  K, B = sheet.take("K4:L35", "isinstance(L,int)")
  A, B = sheet.join("A4:B46704", K, "abs(K - A)",
                dsl.AggregationMethod["minimal"])
  dsl.store_sheet("output.xlsx", "output", [K, L, B])
```

Listing 8. Example Python script for processing Excel data.

the call to `simpleProjection`. In this way, variable parts of the source code and the projection are uniquely connected.

5 Evaluation

5.1 DSL for Spreadsheet Analysis

We implemented a virtual DSL for analysis of Excel spreadsheets. The use case is motivated by a cooperation with researchers from a biomedical field. Their lab uses chromatography methods for genome analysis of viruses. The device performing the analysis exports its results as Excel files which need further analysis to extract relevant data. Due to data variations and lack of programming skills these files are processed manually, which is a repetitive and error-prone task. With the virtual DSL these researchers should be able to use and adapt scripts for automated processing yet retain the flexibility of a host GPL for a future automated processing pipeline.

The implementation uses as the environment JupyterLab and Python as the host GPL. We developed a simple package (library) in Python based on the module *openpyxl* [27] which performs several tasks like loading a spreadsheet, conditional cell selection, joining of cell ranges, and storing the results in a new spreadsheet. Listing 8 shows a typical Python script used in this scenario. Here `dsl` is a qualifier to our package.

For this virtual DSL we implemented four pattern/projection pairs, for the library calls *load*, *take*, *join*, and *store*. Figure 2 illustrates how the code from Listing 8 is actually shown in the JupyterLab hybrid editor.

```
load sheet raw data from Chromatography.xlsx:
    take K, B from K4:L35 where isinstance(L, int)
    join A, B from A4:B46704 on K where abs(K - A) is minimal
    store K, L, B in sheet output of output.xlsx
```

Figure 2. Code from Listing 8 as shown in the hybrid editor.

5.1.1 Discussion.

Implementation of this virtual DSL has shown qualitatively that the proposed concept of a virtual DSL is useful. Moreover, the presented algorithms and the implementation have been verified as correct. The hybrid editor is responsive and due to the special care of input management (Section 4.2.1) offers a smooth editing experience.

However, we did not conduct a rigorous user study. We relied on feedback from one tester as well as our own experiences with the system to conclude high responsiveness and correctness of the implementation. The intended customers have not yet introduced the system (changes in the process require time investment on both sides) but are interested. A more thorough evaluation with a user study remains a part of the future work.

Note that the hybrid editor presentation in Figure 2 is not *radically* more user friendly than the textual representation of code in Listing 8. However, for researchers from non-CS fields the threshold for considering a software solution as 'too complex' is rather low. Consequently, even a moderate improvement of readability and editing experience (also due to parameter recommendations within projections) was relevant in this case.

5.2 DSL for Formula Editing

The next virtual DSL offers graphical formula (or equation) editing using mathematical notations. It demonstrates that our approach can also tackle non-textual projections, see Figure 3. Instead of implementing the desired projections from scratch, we leverage the fact that our hybrid editor can use any HTML/JavaScript components, and build upon Mathlive [28], a web-based graphical editor for mathematical equations. We choose LaTeX as its output format. This enables to use latex2sympy2 [46], a library to transform LaTeX equations into SymPy [12] objects at runtime. SymPy can perform on these objects tasks such as simplification and solving of equations, but it can also generate executable Python functions.

We first implement a small Python library/module `mathdsl` that transforms LaTeX equations into performant (due to internal use of NumPy), executable Python functions. The module provides two functions: `compile` and `evaluate`. Earlier function transforms LaTeX equations into an executable form but does not evaluate them, while the other immediately performs the evaluation.

Both functions assume as their first argument a string containing the LaTeX code. The compile function returns

```
// Pattern for the compile function
expressionPattern`
  ${contextVariable("mathdsl")}
    .compile(${arg("latex", "string")})
`;
// Pattern for the evaluate function
expressionPattern`
  ${contextVariable("mathdsl")}
    .evaluate(${arg("latex", "string")}, locals())
`;
```

Listing 9. Two patterns for the graphical formula editor expressed in the Templating Language.

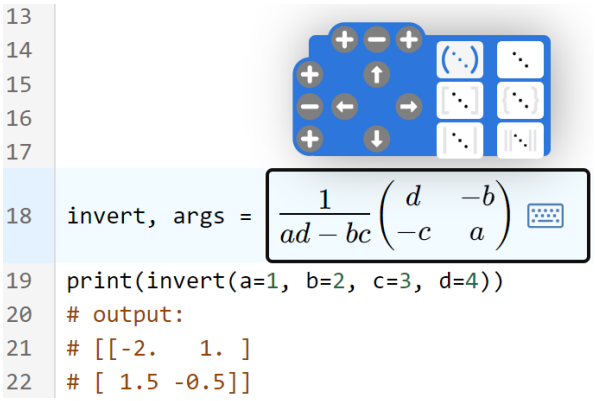

Figure 3. A hybrid editor view during editing a function to compute an inverse of a $2x2$-matrix with `mathdsl`.

the generated function as well as a tuple of free symbols in the equation. For the evaluation function, an additional parameter must be provided next to the code that describes the variable scope to evaluate the term in. The patterns used to match these operations are defined in Listing 9 using the Templating Language. Unlike in previous examples, we declare these patterns to be expressions instead of statements. This makes it possible to embed the equation editor in normal code statements.

The corresponding hybrid editor has been integrated in JupyterLab [40] (see sources of our project) and in the live demo (tab 'Python'). Figure 1 in the introduction shows how the hybrid editor represents a function for rotating a vector. The corresponding Python source code is given in Listing 3. Another example is shown in Figure 3 with the corresponding source code in Listing 10. While editing the formula in the hybrid editor, interactive GUI elements support matrix editing and transformations. Also, visual keyboard is available for entering symbols and operators without knowledge of LaTeX (not shown; available in live demo).

Overall, we conclude that the choice HTML/JavaScript-based projection widgets, allows for a wide range of scenarios and types of projectional forms. Especially this enables access to a large ecosystem of open source libraries and components for web-based GUIs.

```
invert, args = mathdsl.compile("\\frac{1}{ad-bc}
      {\\begin{pmatrix}d & -b \\\\ -c & a \\\\end{pmatrix}}")
print(invert(a=1, b=2, c=3, d=4))
# output:
# [[-2.   1. ]
# [ 1.5 -0.5]]
```

Listing 10. Python source code corresponding to Figure 3.

Table 1. Implementation sizes in non-blank LOC of four operations for the spreadsheed DSL. The total LOC for simplified definition of projections is shown in bold.

Operation	Pattern	Svelte	Svelte (simple)	Total
load	21	49	3	70/**24**
take	22	56	3	78/**25**
join	24	77	3	101/**27**
store	21	55	3	76/**24**

5.3 Analysis of the Development Effort

We analyze and discuss here the implementation size of the virtual DSL from Section 5.1 as a proxy for the development effort of such a DSL. Due to a limited scope of this project we could not yet conducted a controlled developer study on the coding effort. However, such a study can be pursued as a part of the future work.

Table 1 shows the number of non-comment Lines of Code (LOC) the four DSL components. Our original implementation of Svelte components produced a rather verbose code with large LOC values, see column "Svelte" in this table. These large LOC numbers can be significantly reduced by using the simplified definition of projections introduced in Section 4.3. While the simplified definitions slightly limit the flexibility of projections, it is possible to use them for the four DSL components constituting the virtual DSL for analysis of Excel spreadsheets. Listing 7 shows that even the most complex of the 4 projections can be defined in this way. The LOC count becomes for each projection only 3.

On average, the LOC for pattern code is 22 LOC. Using the simplified projection definitions we reach an average of 25 LOC as the implementation length of a DSL component, or 100 LOC for the whole spreadsheet DSL. This is already comparable to the size to the Python library code (69 LOC). The original 'naive' implementation of Svelte components gives an average of 59.25 LOC per projection and average of 81.25 LOC per DSL component. We conclude that the simplified definition of projection templates can significantly reduce the implementation effort and improve maintability.

5.4 Generating Patterns and Projections from Samples: Advantages and Limitations

The approach for generating DSL components from examples introduced in Section 3 might even further reduce the development effort and make virtual DSL accessible for users

```
# Code sample 1
with dsl.load_sheet("chroma.xlsx", "raw data") as sheet:
  sheet.do_something()
# Projection sample 1
load sheet raw data from chroma.xlsx :

# Code sample 2
with excel_dsl.load_sheet("cost.xlsx", "April") as sheet:
  sheet.do_some_other_thing()
# Projection sample 2
load sheet April from cost.xlsx :
```

Listing 11. Samples for the *load* operation.

with limited programming skills. The majority of the code required for defining patterns and projections can be provided by the generator, thus decreasing the initial hurdle for developers to create a projectional editor for their DSL. The key question is whether the generator can create correct patterns and projection templates for realistic scenarios. To this end, we attempted to generate these components for all DSL components of the spreadsheet DSL from Section 5.1. Due to space limits, we describe here only the results for the *load* and *take* operations (results for *join* and *store* were similar).

Running the samples from Listing 11 through our generator algorithm yields a pattern and projection template whose placeholders are correctly connected. Note the space character before the colon in the projection samples, which is required to satisfy the tokenizer used in the generator.

This evaluation reveals some shortcomings of the approach from Section 3.1. In the chosen samples, the nested statements included in the respective `with` statements' blocks only differ by the method identifier after the dot infix. Thus, the algorithm generates a pattern that includes two variable identifier nodes at these positions instead of a block. If we replace these two nested statements by code without any AST overlap, such as `x = 42` and `print("Hello")`, then a wildcard placeholder is generated which would match an arbitrary statement, but not a block with multiple statements. To express the intent of including multiple statements in the block, one more statement must be added to one of the samples. Now, the algorithm correctly detects a block at this position in the AST.

Furthermore, the generator does not perform any kind of context analysis. Thus, the variable nodes `dsl` and `excel_dsl` are not identified as context variables but simply as placeholder nodes of type *identifier*. Similarly, the generated pattern does not provide any context variables to the nested block. These two parts must be edited by hand in the generated pattern.

Finally, it is advisable to change the generated variable names to more meaningful names to help with later maintenance. We call the two variables that map to parts of the projection `fileName` and `sheetName`. The context variable

```
# Pattern
with ${contextVariable("dsl")}.load_sheet(
  ${arg("fileName", "string")},
  ${arg("sheetName", "string")}
) as sheet:
  ${block({ sheet: "sheet" })}

# Projection
load sheet sheetName from fileName :
```

Listing 12. Final pattern and projection for the *load* operation.

```
# Code sample 1
K, L = sheet.take("K4:L35", "isinstance(L, int)")
# Projection sample 1
take K, L from K4:L35 where isinstance(L, int)

# Code sample 2
A, B, C = the_sheet.take("A1:C100", "A + B == C")
# Projection sample 2
take A, B, C from A1:C100 where A + B == C
```

Listing 13. Samples for the *take* operation.

for the DSL module name is simply called `dsl` and the sheet context variable passed to the nested block becomes `sheet`.

The final pattern and projection are shown in Listing 12. For sake of brevity, we show only a textual form of the projection instead of the code of the generated Svelte component.

Listing 13 shows samples for the operation *take*. Similar problems with *load* occur here at the attempt to generate a pattern and a projection template. Instead of a problem with detection of a block, the code samples must have a different amount of identifiers before the assignment to express that the pattern should not match the identifiers themselves as placeholders but the whole list of identifiers. Also here context variables are incorrectly detected and must be replaced by the context variable `sheet` in the generated pattern. As before, we rename the variables to be more meaningful.

This case study revealed how our generator can be further improved. Context variables in particular are not covered by the generator and require manual editing of the generated code. Currently, the code samples provided to the generator are isolated per DSL operation, preventing a connection of placeholders that appear in multiple operations. Comparing the code samples of different operations should allow the generator to identify placeholder nodes as context variables if they appear in multiple operations but not in any of the projections. Furthermore, the declarations of nested blocks could automatically be filled with the context variables by analyzing the identifier or parameter nodes affecting the variable scope inside the block. This however requires knowledge about the host language's scope resolution behavior.

6 Related Work

Research areas relevant to our work are *hybrid textual editors*, *projectional editors*, *Domain Specific Languages*, and *language workbenches*. Furthermore, work in *programming by example* is connected to our code generation approach.

Hybrid textual editors relate directly to our work. We use this term to bracket contributions which extend textual code editors with the ability to express parts of the code in an alternative form, typically as visual GUI elements. Contrary to our work, the approaches described below do not address generation of such GUI components from examples.

Eisenberg and Kiczales [18–20] propose *ETMOP*, an Eclipse-based editor which renders selected parts of Java code as interactive graphical elements. Using Java 5 annotations, a developer specifies which methods and classes should be shown in a graphical way. Similarly to our work, ETMOP uses pattern matchers to recognize these annotations. However, our approach matches directly the code parts to be shown as projections and does not require annotations or other developer intervention. Our technique also works for host GPL languages which do not support annotations.

Later works [1, 54, 59] share the same concept of specifying which parts of code are represented in an alternative form via annotations or syntax extension of the edited code.

Work of Renggli *et al.* propose *language boxes* [59], a modular mechanism to encapsulate language extensions such as DSLs. The key idea is to extend the grammar of the host language (here: Smalltalk) using the concept of executable grammars [5]. The approach can combine different textual notations using delimiters inserted explicitly using a special-purpose editor. It assumes that the grammar, compiler and editor of the host language can be extended, which incurs a considerable effort for mainstream languages like Python.

Andersen *et al.* propose syntactic extensions for the Racket language [24] in terms of *interactive syntax* [1]. Such (textual) syntax extensions are rendered as interactive GUIs by an appropriate editor, here DrRacket. The authors posit that the approach is compatible with other languages with a macro system, such as Closure, Julia, Rust, or even C++. Contrary to this, our method does not require a macro system and works with languages like Python or JavaScript. Our approach also supports compositionality, i.e projections can be nested.

Omar *et al.* introduce live literals, or *livelits* which embed user-defined GUIs into textual code [54]. The implementation in Hazel/OCaml extends the above-cited work [1] with compositionality, type safety, and liveness. The latter means that the evaluation of livelits occur in the runtime environment of the program being written. As [1], livelits require a host language with a macro system and a pure (functional) host language, differently to our work. However, the authors note that imperative languages can be also covered, with more effort.

A slightly different flavor of a hybrid textual editor is described in [55]. Upon triggering the code completion functionality of the editor, the developer is presented with a context-sensitive palette, i.e. a GUI which allows for an interactive specification of expressions, statements or API parameters. A prototype called *Graphite* provides palettes for regular expressions and for color selection in Java. Differently to our approach, palettes must be associated with a target class via annotations or explicitly via menus. Also, Graphite shows the enriched code representation only locally and during editing, which limits code comprehension.

Barista [41] is an implementation framework designed for creating projectional editors with a strong support for textual input. Editors constructed with Barista represent code as a proprietary data structure (a model), but they deploy parsing techniques that treat the structure as if it were textual. For example, equations are displayed visually as math, but when a caret is placed on them, a textual view is shown.

French *et al.* [26] propose a similar system which can embed interactive graphical objects in Java or Python textual code. The internal representation takes form of a modified AST. Contrary to our work, the last two approaches must be aware of all syntactic structures of a target language, leading to a considerably larger development effort.

Further systems add visual programming to textual code. Examples include Boxer [17], Scratch [60], or Smalltalk with its live programming capability [48]. Mathematica [35] supports expressions which include visual elements such as images or diagrams. These can copied and pasted into textual expressions but also manipulated in a GUI-fashion. Jupyter and JupyterLab [40] allow inserting widgets like sliders that modify parameters of code. The Jupyter API extension *mage* [39] enables creation of tools that can represent themselves as both code and GUI as needed.

Our previous work [2] enhances text editing by embedding user-defined DSLs as code comments. During editing, a code completion action on these comments expands a DSL expression into regular Python or R code. Our tool *NLDSL* [10] is implemented as an extension for Visual Studio Code and can be easily ported to all IDEs which support Microsoft's Language Server Protocol [13]. Contrary to work presented here, NLDSL requires developers to edit DSL expressions and manually trigger code generation. Moreover, NLDSL supports only textual DSLs.

Projectional editors were pioneered in the 1970s and 1980s by projects like *Cornell Program Synthesizer* [65], *Incremental Programming Environment* [49], and *GANDALF* [52]. A key feature of such systems is the ability for provide different ways of viewing and editing program components. For example, embedded data can be shows as a spreadsheet, state machine as a diagram, and other parts of code as text. In particular, they can express code fragments in a simplified form resembling a DSL. Projectional editors store and manipulate programs directly as an abstract syntax tree, so

parsing is not necessary [9, 15, 37]. Consequently, they can mix arbitrary programming languages (multiple GPLs and DSLs) in a single view without ambiguity.

While text-based languages can be edited in any text editor, projectional editing ties the language to a specialized editor that is aware of the language's syntax and possibly semantics [66, Chap. 7.3]. Voelter *et al.* [67] conducts an analysis of user-friendliness of projectional editing based on JetBrains MPS. Their conclusion is that projectional editors do not lead to code being written faster than in conventional text editors. The authors identify usability challenges in the areas of (i) efficiently entering textual code, (ii) selecting and modifying code, and (iii) infrastructure integration.

Approaches such as *grammar cells* [68] address the challenges (i) and (ii) and attempt to improve editing experience in such editors. The key idea is a formalism for textual notations via declarative specification of the projectional editor's behavior. In our work we use a sophisticated input management via focus managers (Section 4.2.1) to improve the editing experience in the projections.

Domain Specific Languages (DSLs) [16, 25, 38, 43, 66] focus on a limited domain but allow its modeling in a concise and readable way. By 'making it hard to do something wrong' [25], they can also reduce the amount of defects and improve maintainability.

Research activities related to DSLs include extensible languages like Racket [23, 24], language-oriented programming and language workbenches [31, 44], analysis of the usage and properties of DSLs [25, 43, 50], syntactic macros [4, 24, 34], or enhancing libraries by "syntactic sugar" [22].

Modern DSL engineering frameworks like textX [16] or Spoofax [38] significantly lower the cost of developing a DSLs. An even higher level of productivity and toolchain integration can be achieved by language workbenches discussed below.

Language workbenches such as Jetbrains' MPS [7], MontiCore [44, 61], or Xtext [3] complement and extend DSL development frameworks by providing editors with syntax checking and code completion for created DSL or GPL languages. They also facilitate code parsing and generation.

JetBrain's *Meta-Programming System (MPS)* [7, 36, 56] is probably the most popular publicly available language workbench. Since it uses projectional editing, users can switch between different notations or visualizations of the same program. The AST manipulated by the editor is stored in form of an XML file. It is then translated into a target language such as Java or C by the MPS code generator.

Lafontant *et al.* propose *Gentleman*, a lightweight Web-based language workbench [45]. It offers commonly used interface layouts, such as tables, or horizontal and vertical stacks to design custom visual representations of code. It is similar to our approach by using web-based components as projections and their styling through CSS.

Contrary to our approach, these systems persist source code in a proprietary format. This makes it harder to use other editors and complicates integration with existing tools.

Programming by example (PBE) is a form of program synthesis which uses input and/or output examples to specify the desired code, typically a DSL expression [6, 21, 30, 63, 64]. In the past decade, many sophisticated approaches have been developed, including FlashMeta [57], a framework for designing and implementing program synthesis engines for custom DSLs. PBE has been commercially exploited in e.g. Excel, PowerShell and other products to automate string manipulation, data preprocessing, and other tasks [29, 30, 47, 58]. Recently introduced Large Language Models have also been applied in the field of program synthesis, typically in the context of input specification by natural language [53].

Our method for generating patterns and projection templates from examples (Section 3) is a form of highly specialized PBE. However, instead of a DSL we generate domain-specific GPL code. For patterns, the algorithm described in Section 3 creates an AST-based data structure which is then converted into a Template Language, i.e. annotated JavaScript code. For projections, we use code skeletons which are populated with the corresponding placeholders (see Section 3.4). Compared to work referenced above, our approach is more narrow and less flexible but has lower complexity and implementation effort.

7 Conclusions and Future Work

We proposed an approach and a tool for hybrid editing of code by mixing a textual representation and projectional views. Latter can assume virtually any form - from textual, DSL-like formats to tables to equations. The editor directly manipulates the source code of the host GPL. This enables seamless integration of the virtual DSL and the GPL, and facilitates integration in the development toolchain. Our evaluation shows that the approach provides a good developer's experience and requires an acceptable effort of developing DSL components. To reduce this effort, we have proposed an approach to generate virtual DSL components from samples of code and corresponding textual projection.

Our future work will include an implementation of a larger use scenario in order to refine the approach and the evaluation. Another task is to offer our hybrid editor as an extension for Visual Studio Code. We will also improve the approach for generating DSL components from samples by cross-component analysis as discussed in Section 5.3. An interesting but rather challenging problem is generation of non-textual projections from examples. Furthermore, a more rigorous analysis of the soundness and completeness of the proposed algorithms is needed. Finally, adding type checking for projectional editing could improve user experience.

References

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (nov 2020), 28 pages. https://doi.org/10.1145/3428290

[2] Artur Andrzejak, Kevin Kiefer, Diego Elias Costa, and Oliver Wenz. 2019. Agile Construction of Data Science DSLs (Tool Demo). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Athens Greece). ACM, 27–33. https://doi.org/10.1145/3357765.3359516

[3] Lorenzo Bettini. 2016. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition* (2nd ed.). Packt Publishing.

[4] Jeffrey Werner Bezanson. 2015. *Abstraction in technical computing [Julia language].* Thesis. Massachusetts Institute of Technology. http://dspace.mit.edu/handle/1721.1/99811

[5] Gilad Bracha. 2007. Executable Grammars in Newspeak. *Electronic Notes in Theoretical Computer Science* 193 (2007), 3–18. https://doi.org/10.1016/j.entcs.2007.10.004

[6] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (jan 2023), 30 pages. https://doi.org/10.1145/3571226

[7] Fabien Campagne. 2016. *The MPS Language Workbench Volume I: The Meta Programming System (Volume 1)* (3rd ed.). CreateSpace Independent Publishing Platform, USA.

[8] Sergei Chestakov. 2022. *Betting on CodeMirror.* Retrieved September 10, 2023 from https://blog.replit.com/codemirror

[9] Tony Clark. 2015. A General Architecture for Heterogeneous Language Engineering and Projectional Editor Support. (2015). arXiv:1506.03398 [cs]

[10] NLDSL contributors. 2023. NLDSL Overview. https://aip.ifi.uni-heidelberg.de/software/nldsl Accessed on September 10, 2023.

[11] Svelte contributors. 2023. Svelte - Cybernetically enhanced web apps. https://svelte.dev/ Accessed on September 10, 2023.

[12] SymPy contributors. 2023. SymPy - a Python library for symbolic mathematics. https://www.sympy.org Accessed on September 10, 2023.

[13] Microsoft Corp. 2023. Language Server Protocol Specification. Retrieved September 10, 2023 from https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[14] Microsoft Corp. 2023. Monaco - The Editor of the Web. https://microsoft.github.io/monaco-editor/ Accessed on September 10, 2023.

[15] Riwan Cuinat, Ciprian Teodorov, and Joel Champeau. 2020. SpecEdit: Projectional Editing for TLA+ Specifications. In *2020 IEEE Workshop on Formal Requirements (FORMREQ).* 1–7. https://doi.org/10.1109/FORMREQ51202.2020.00008

[16] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. 2017. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems* 115 (Jan. 2017), 1–4. https://doi.org/10.1016/j.knosys.2016.10.023

[17] A. A diSessa and H. Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Commun. ACM* 29, 9 (sep 1986), 859–868. https://doi.org/10.1145/6592.6595

[18] Andrew David Eisenberg. 2008. *Presentation techniques for more expressive programs.* Ph. D. Dissertation. University of British Columbia. https://doi.org/10.14288/1.0051292

[19] Andrew D. Eisenberg and Gregor Kiczales. 2006. A Simple Edit-Time Metaobject Protocol: Controlling the Display of Metadata in Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06).* Association for Computing Machinery, New York, NY, USA, 696–697. https://doi.org/10.1145/1176617.1176679

[20] Andrew D. Eisenberg and Gregor Kiczales. 2007. Expressive Programs through Presentation Extension. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development* (Vancouver, British Columbia, Canada) *(AOSD '07).* Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1218563.1218573

[21] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021).* Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

[22] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11).* ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

[23] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

[24] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (feb 2018), 62–71. https://doi.org/10.1145/3127323

[25] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

[26] G.W. French, J.R. Kennaway, and A.M. Day. 2014. Programs as Visual, Interactive Documents. *Softw. Pract. Exper.* 44, 8 (aug 2014), 911–930. https://doi.org/10.1002/spe.2182

[27] Eric Gazoni and Charlie Clark. 2023. openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files. https://openpyxl.readthedocs.io/en/stable/ Accessed on September 10, 2023.

[28] Arno Gourdol. 2023. Mathlive - Equations served with a side of interaction. https://cortexjs.io/mathlive/ Accessed on September 10, 2023.

[29] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *SIGPLAN Not.* 46, 1, 317–330. https://doi.org/10.1145/1925844.1926423

[30] Sumit Gulwani. 2016. Programming by Examples (and its Applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems.* IOS Press. https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/

[31] Gopal Gupta. 2015. Language-based Software Engineering. *Sci. Comput. Program.* 97, P1 (Jan. 2015), 37–40. https://doi.org/10.1016/j.scico.2014.02.010

[32] Jordan Harband, Shu-yu Guo, Michael Ficarra, and Kevin Gibbons (Eds.). 2020. *ECMAScript® 2021 Language Specification* (12 ed.). Ecma International.

[33] Marijn Haverbeke. 2022. *CodeMirror Decoration Example.* Retrieved September 10, 2023 from https://codemirror.net/examples/decoration/

[34] David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *Programming Languages and Systems*, Sophia Drossopoulou (Ed.). Lecture Notes in Computer Science, Vol. 4960. Springer Berlin Heidelberg, 48–62. https://doi.org/10.1007/978-3-540-78739-6_4

[35] Wolfram Research Inc. 2008. *Dynamic Interactivity – Wolfram Mathematica Tutorial Collection.* Wolfram Research Inc. https://library.

wolfram.com/infocenter/Books/8513/

[36] JetBrainsTV. 2017. *Why JetBrains MPS.* Retrieved September 10, 2023 from https://www.youtube.com/watch?v=XGm_khXZl44

[37] Ján Juhár and Liberios Vokorokos. 2015. A Review of Source Code Projections in Integrated Development Environments. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 923–927. https://doi.org/10.15439/2015F289

[38] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497

[39] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

[40] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[41] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*. Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1124772.1124831

[42] Niklas Korz. 2022. *Projectional Editing of Internal Domain-Specific Languages.* Master's thesis. Heidelberg University.

[43] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (March 2016), 77–91. https://doi.org/10.1016/j.infsof.2015.11.001

[44] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *Int. J. Softw. Tools Technol. Transf.* 12, 5 (sep 2010), 353–372. https://doi.org/10.1007/s10009-010-0142-1

[45] Louis-Edouard Lafontant and Eugene Syriani. 2020. Gentleman: A Light-Weight Web-Based Projectional Editor Generator. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (New York, NY, USA) *(MODELS '20)*. Association for Computing Machinery, 1–5. https://doi.org/10.1145/3417990.3421998

[46] latex2sympy2 contributors. 2023. latex2sympy2 - Parse LaTeX math expressions. https://github.com/OrangeX4/latex2sympy Accessed on September 10, 2023.

[47] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

[48] John Maloney, Kimberly M. Rose, and Walt Disney Imagineering. 2001. An Introduction to Morphic: The Squeak User Interface Framework. In *Squeak: Open Personal Computing and Multimedia*, Mark Guzdial and Kimberly Rose (Eds.). Prentice Hall, 39–77.

[49] R. Medina-Mora and P.H. Feiler. 1981. An Incremental Programming Environment. *IEEE Transactions on Software Engineering* SE-7, 5 (1981), 472–482. https://doi.org/10.1109/TSE.1981.231109

[50] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *Comput. Surveys* 37, 4 (dec 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[51] Eugene W. Myers. 1986. AnO(ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (nov 1986), 251–266. https://doi.org/10.1007/bf01840446

[52] David Notkin. 1985. The GANDALF project. *Journal of Systems and Software* 5, 2 (1985), 91–105. https://doi.org/10.1016/0164-1212(85)90011-1

[53] Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[54] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059

[55] Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. https://doi.org/10.1109/icse.2012.6227133

[56] Vaclav Pech, Alex Shatalin, and Markus Voelter. 2013. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, Martin Plümicke and Walter Binder (Eds.). ACM, 165–168. https://doi.org/10.1145/2500828.2500846

[57] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

[58] Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 882–890. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15034

[59] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. 2010. Language Boxes. In *Software Language Engineering*, Mark van den Brand, Dragan Gašević, and Jeff Gray (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–293.

[60] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[61] Bernhard Rumpe, Katrin Hölldobler, and Oliver Kautz. May 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021* (2021 ed.). Shaker Verlag. https://www.se-rwth.de/research/MontiCore/ Aachener Informatik-Berichte, Software Engineering, Band 48.

[62] Tree sitter contributors. 2023. Tree-sitter - a parser generator tool and an incremental parsing library. https://tree-sitter.github.io/tree-sitter/ Accessed on September 10, 2023.

[63] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 326–340. https://doi.org/10.1145/2908080.2908102

[64] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* PhD Thesis. University of California at Berkeley, Berkeley, CA, USA.

[65] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (sep 1981), 563–573. https://doi.org/10.1145/358746.358755

[66] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[67] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Lecture Notes in Computer Science, Vol. 8706. Springer International Publishing, 41–61. https://doi.org/10.1007/978-3-319-11245-9_3

[68] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365

Received 2023-07-14; accepted 2023-09-03