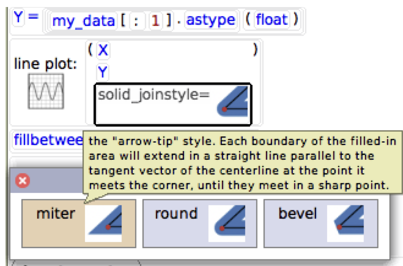
Visual Replacements: Cross-Language Domain-Specific Representations in Structured Editors

Tom Beckmann
tom.beckmann@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

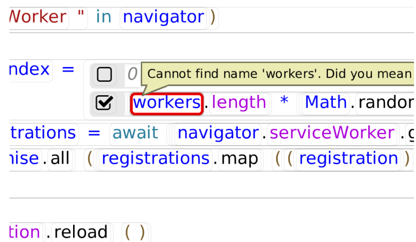
Daniel Stachnik
daniel.stachnik@student.hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Jens Lincke
jens.lincke@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

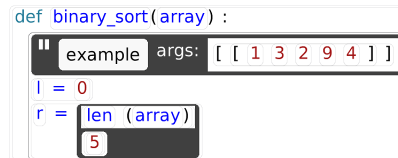
Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany



(a) Python matplotlib: a string mapped to allow choosing from a menu.



(b) A toggle in TypeScript: diagnostics work in the expressions nested in visual replacements.



(c) Babylonian Programming in Python: an example and a probe.

Figure 1. Visual replacements work across languages and integrate with language support to enable use of domain-specific representations (DSRs) in a wide range of use cases.

Abstract

To help developers work at the level of abstraction of their choice, some editors offer to replace parts of source code with domain-specific representations (DSRs). Typically, such replacements are language-specific and optimized to limited use-cases. In this paper, we extend a general-purpose structured editor with *visual replacements*, a mechanism to replace code of any programming language with DSRs. A *visual replacement* consists of language-dependent queries to map arbitrary syntax trees and a language-independent DSR of differing abstraction and interactivity, ranging from simple images to graphical user interfaces that modify source code. Our extension synchronizes source code and DSR automatically, while ensuring that language support such as autocompletion and error checking work even inside a replacement. We demonstrate the use and applicability of the replacement mechanism in three case studies.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; *Application specific development environments.*

Keywords: domain-specific representation, visual replacements, structured editing

This work is licensed under a Creative Commons Attribution 4.0 International License.

PAINT '23, October 23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0399-7/23/10.
https://doi.org/10.1145/3623504.3623569

ACM Reference Format:
Tom Beckmann, Daniel Stachnik, Jens Lincke, and Robert Hirschfeld. 2023. Visual Replacements: Cross-Language Domain-Specific Representations in Structured Editors. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '23), October 23, 2023, Cascais, Portugal*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3623504.3623569

1 Introduction

Developers seek different means to express code depending on the task at hand. As a tendency, the closer the match between the task's domain and their means of expression, the fewer workarounds that may introduce faults or mismatches are required to formulate code for the task. Some tasks have inherent visual properties, such as choosing a color or configuring a visualization. Typically, however, these tasks are mapped to a textual interface within integrated development environments, posing a mismatch against a natural formulation in the task domain that would show visual properties. Within this paper, we refer to *widgets* as user interface elements that replace, wrap, or annotate the default view on source code and instead provide a domain-specific representation (DSR). This is independent of whether the user is working with a structured editor or textual source code.

A number of works have demonstrated the possibilities of editors that augment source code, to maintain the powerful combination and abstraction mechanisms found in programming languages, with widgets to provide a closer match to a task's domain [1, 9, 12, 13, 16]. However, these approaches are designed for particular languages and often even require special syntactic extensions, hindering their adoption in existing programming language ecosystems, particularly with regards to the language's tool ecosystem.

In this paper, we introduce a concept of *visual replacements* that work across languages and wrap or replace arbitrary syntax trees. In the presence of compatible semantics in targeted languages, a single visual replacement can even be reused across these languages with minimal effort. A language gains support through a *runtime mapping* that informs the visual replacements how expressions can be evaluated and, for each visual replacement, a mapping from that language's syntactic concepts to the data the replacement needs to construct its user interface, as shown in Figure 2. Visual replacements are designed to make use of built-in syntactic constructs to integrate with general-purpose languages without interfering with their function in their existing ecosystem of tools.

In the following, we will first discuss related work to our concept. We then derive a set of requirements for a visual replacement concept, before describing the concept itself. We will then describe case studies of its use using the example of our reference implementation of visual replacements in the structured editor Sandblocks [2]. Finally, we discuss its applicability and limitations, future extensions, and conclude the paper.

2 Related Work

A number of approaches exist that extend a language with specific types of syntactic elements or annotations to provide widgets.

Graphite is an implementation of active code completion [13], which augments autocompletion in IDEs with widgets defined in palettes for specific types for the Java language. Users invoke autocompletion on a type for which a palette is defined to open a widget for the expression's text region. After completing an editing task with a widget, the text region is replaced with the widget's textual result. A pattern explored in Graphite is to store additional data that a widget generates in a comment adjacent to the source code.

Interactive syntax [1] describes a new syntactic element in the Dr. Racket programming environment that extends the editor to support widgets. The special syntactic element denotes both the runtime effect of the element and stores additional data that the editor may need.

Livelits are a mechanism to offer composable, widgets in the Hazel programming environment [12]. To use a livelit widget, programmers type out a reference to a prior livelit definition, which then displays a widget next to the reference. Unlike the previous two approaches, livelit are designed to contain nested editors for Hazel expressions that themselves can contain livelits.

Projectional editors are designed to be able to provide different projections on a part of a program. The programming environment mbeddr [16] for example allows users to edit state machines using a widget, or mathematical expressions using a notation that breaks from the linear layout imposed by most textual editors. Similarly, the structured editor Barista [10] allows programmers to define custom views that compose with other views defined in Barista. Both editors store syntax trees in a custom serialized format that allows to seamlessly compose languages and store any data needed by the projections.

Other approaches detect patterns as they exist in regular source code and show widgets nearby or instead of that source code. Implementations range from visualizing the effect of the source code to allowing edit operations that are written back to code.

Examples are most commonly available text editors, such as those based on the JavaScript editor framework CodeMirror [7]. Through a matching query, developers can for example specify that hexadecimal color codes should show a color picker next to them.

Moonchild [5] demonstrates how widgets can replace regions of source code. For example, a comment containing markdown syntax is replaced by a rich-text editor that shows the rendered markdown. Extra data needed for the function of widgets is stored in comments as well.

3 Requirements

Visual replacements are designed to work across languages and to integrate well with existing ecosystems: a mechanism that integrates with and modifies source code necessarily coexists with a variety of other tools. Especially for text-based languages, a number of tools, such as version control, global text-based search, and language tooling, for example via the

Language Server Protocol, have become indispensable in the workflows of many programmers. Visual replacements target the same ecosystem and as such need to follow some practical constraints to fit in well. We decided on the below list of requirements after multiple iterations of the design and alternative approaches, balancing different trade-offs.

No new syntax. Adding new syntax would require adapting parsers of all tools that depend on a language's syntax, such as compilers and language support. In some ecosystems, such as Racket or JavaScript, extending and modifying the syntax is comparatively well-supported but even then it requires all tools to be aware of the syntax extensions.

Graceful degradation. Editors that do not support displaying visual replacements will display a serialized form, ideally in the host language's syntax of the visual replacement. Visual replacements should aim to look native to the conventions of the textual form of the host language to allow working with their serialized form without damaging it. For example, when merging changes to the same visual replacement in a version control system, it should be obvious how parts of the replacement's data should be reassembled when a conflict occurs.

Avoid comments. While comments in textual programming languages would serve well to add arbitrary data, they are also not considered during error checking or automated refactoring operations. As visual replacements support nesting expressions, preserving language support within a visual replacement's data is an important factor. Additionally, careless merging or accidental editing to the serialized format in a comment would yield errors only once opened in a programming environment supporting visual replacements, while use of the host language's syntax ensures that the existing language tooling can communicate violation of some constraints.

Static and dynamic information. Visual replacements should be able to make use of both static information, found in the source code's syntax tree, and dynamic information obtained from a runtime. To that end, adding support for visual replacement in a new language should entail finding ways of allowing execution of code snippets in a context and reporting results to the replacement in a serialized form.

Open ceiling. Visual replacements should augment functionality accessible through source code and not block users' abilities to express complex constructs. For example, while it may be nice to create a visual replacement that contains a slider to set some values, ideally that slider could also be replaced by a variable. When designing visual replacements, authors should thus prefer to nest different visual replacements rather than create monolithic visual replacements that bundle and hide functionality. Unlike other visual

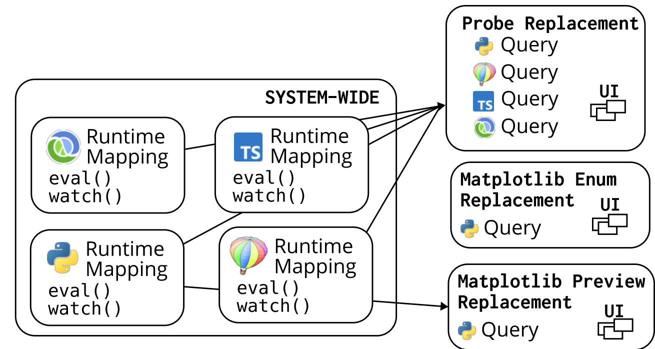


Figure 2. Visual replacements integrate with the source code of existing languages. They require the definition of a runtime mapping that exposes an evaluate and a watch primitive once per language, as well as the definition of a query per language per replacement. The user interface definition of a replacement, the DSR, can be shared across languages.

approaches, users should always be able to show visual replacements as plain source code, as opposed to the DSR.

4 Replacement Mechanism

Source code is at some point in its lifecycle represented as a tree of syntax nodes, its abstract syntax tree (AST). For textual languages, this process occurs during parsing. In structured editing, the source code is edited at the AST-level directly, skipping parsing.

Our visual replacement concept builds on top a continuously available representation of the AST, as such it benefits from the use of a structured editor. To obtain a structured editor for arbitrary textual languages, an approach such as Sandblocks may be used [2], allowing users to generate a structured editor from a grammar for a textual programming language. A continuously available syntax tree that is stable across arbitrary textual edits can be approximated in text editors through a set of heuristics that match syntactic constructs based on their locations between edits.

The syntactic elements that a language is composed of serve different functions: some express runtime behavior, some assert assumptions or properties on state or data flow, some facilitate abstraction. Depending on the aspect a visual replacement is designed to support, any of these may be replaced or wrapped with a visual replacement.

4.1 Visual Replacements

Authors of visual replacements work with two data types: the visual replacement itself and a runtime mapping, if information from a runtime is required for the replacement, as illustrated in Figure 2. Conceptually, a visual replacement is made up of a constructor for the replacement's user interface, one or many language-dependent matching queries,

and an optional template with an accompanying keyword for invocation.

```
VisualReplacement :=
  build: (bindings: Map<string, AST>) => UI
  queries: {language: symbol, query: string}
  template: {keyword: string, code: string}
```

For obtaining dynamic information, visual replacements need a runtime mapping.

```
RuntimeMapping :=
  language: string
  watch: (node: AST) => AST
  eval: (node: AST) => Object
```

We will explain the function of the parts of a visual replacement in subsection 4.3 and of a runtime mapping in subsubsection 4.3.5.

4.2 Types of Visual Replacements

We distinguish between different types of visual replacements along three factors, depending on the way they integrate with the syntactic constructs and runtime of a language. These factors demonstrate the design space for authors of visual replacements.

First, visual replacements may either be *static* or *dynamic*. A dynamic visual replacement reports runtime information to the replacement within the editor, which the replacement then may make use of to display information. A static replacement only displays statically available information, such as information that can be derived from the syntax tree or additional documentation.

Second, visual replacements may either be *terminal* or *non-terminal*. A terminal replacement replaces a subtree and nests no other syntactic nodes inside it. However, it may contain text fields, images or other user interface elements. A non-terminal replacement also replaces a subtree but nests at least one syntactic node inside it. Consequently, further visual replacements may occur within a non-terminal replacement, if the nested syntactic nodes match another replacement.

Examples for static, terminal visual replacements are simple data editors, as shown in Figure 3. For example, programs may be augmented with sliders, checkboxes, or dropdowns to make editing parts of a program more direct and less reliant on a keyboard. This may be particularly beneficial in a part of a program that defines a static data structure.

Static, non-terminal visual replacements are well-suited, for example, for dynamic data structures that still have an inherent visual structure. A state machine object or array that defines a table lend themselves well to be represented with a visual structure instead of source code but both may nest arbitrary expressions. For an example from related work, consider *interactive syntax*[1], where the representation in source code of a red-black tree is replaced with a dynamic graph visualization of a red-black tree. As such, the visual replacement makes the inherent visual structure of the data

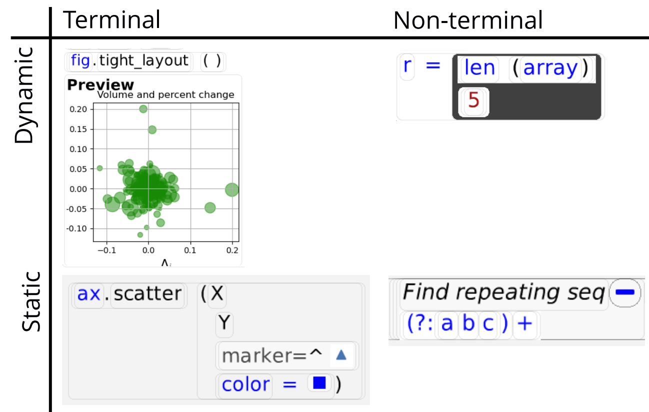


Figure 3. Examples for each combination of static/dynamic and terminal/non-terminal. Top left: a preview for a plot, replacing a `plt.show()` call; top right: a Babylonian probe; bottom left: an enum with icon previews and a color picker; bottom right: a comment around a part of a regular expression.

structure visible but still exposes the potentially non-visual aspects: expressions calculating or defining its data fields as expressions in the original language.

Dynamic, terminal visual replacements can be used for visualizations. A "save to file" method call may be replaced in its entirety by view on that file's contents.

Dynamic, non-terminal visual replacements are useful to wrap expressions that are evaluated at runtime and the result of the evaluation fed back to the programming environment. This allows modeling an inline probe to visualize the runtime value of an expression or a memoization replacement.

A third factor concerns the way replacements are stored and thus also restored after a file has been closed and reopened. Here, visual replacements may either be *implict* or *explicit*. For explicit replacements, the replacement adds a marker to the AST. This could be an annotation, comment, or other means to inform the system that the user explicitly opted to display this AST node as visual replacement. In contrast to this, implicit visual replacements match against source code as it would be written without visual replacements in mind. There is still a specific rule to find whether the replacement applies here but no traces of the visual replacement are stored in the AST, thus also not revealing whether the user did opt-in to a replacement explicitly.

4.3 Lifecycle of Visual Replacements

The lifecycle of visual replacement can be regarded in separate steps. First, some trigger invokes the system to consider an AST node for replacement. The system then traverses the list of replacements to match eligible options. A matched option is then constructed and replaces the AST node. These steps are visualized in Figure 4. Subsequent edits to the visual

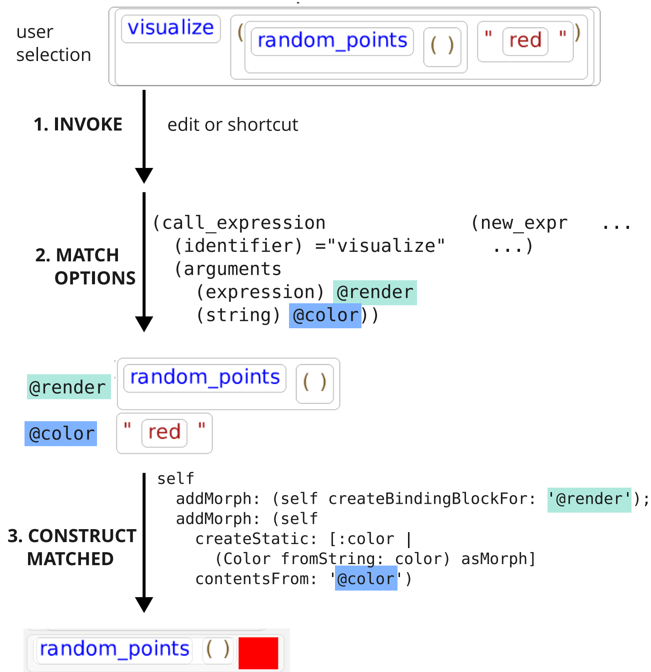


Figure 4. Lifecycle of a visual replacement from its invocation to eventual display in the editor.

replacement are synced to the underlying AST. The visual replacement can interface with the runtime, if needed.

4.3.1 Invoke. Our reference implementation of visual replacements supports three means of invoking the replacement mechanism. First, the user selects an AST node and presses a dedicated, configurable shortcut. The system then moves to the next step for the selected node, trying to locate a suitable visual replacement as described below. In practice, this explicit invocation is most often used to invoke *implicit* replacements, those without a prior explicit marking as replacement in the AST. As part of being invoked, the replacement's implementation may also choose to rewrite the AST to add a marker, thus making the visual replacement explicit.

Second, a replacement may define a template and keyword that is used to invoke it. As the user begins typing identifiers in any syntax node, visual replacements whose keyword matches the identifier are suggested to the user for invocation in an autocompletion menu, as shown in Figure 5. The provided template is used to instantiate the AST that the visual replacement will be matching. This functionality is available to both explicit and implicit replacements. Similarly, authors of visual replacements can define palettes similar to Scratch's or Snap's block palettes [6, 15], containing template visual replacements that users can place in their program via drag-and-drop.

Third, as explicit replacements have specifically been marked by the user as such, we invoke *explicit* replacements

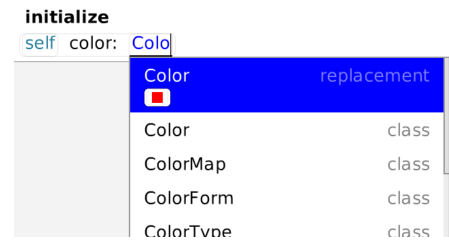


Figure 5. The user has started typing "color" in a Smalltalk method. The programming environment offers both the regular autocompletion entries from language support tools, as well as a visual replacement that has "color" declared as its keyword.

automatically. Here, the system recursively traverses the AST upon its initial load, as well as during edits to the AST, and checks whether any visual replacements marked as explicit would match the given node, as described below.

4.3.2 Match Options. Visual replacements define a query to match against the structure of a node's syntax subtree. Queries in our reference implementation use an extended form of Tree-sitter's query mechanism [3].

In the below example, the first form in each S-Expression is used to denote types of AST nodes. Subsequent forms are used to describe nested children in the tree and their properties. The equal sign is used to match against the textual contents of the form preceding it. The "@" sign is used to put a matched node in a binding for later use during construction.

```
(call_expression
  (identifier) ="visualize"
  (arguments
    (expression) @render
    (string) @color))
```

The example thus matches expressions in source code such as `visualize(random_points(), "red")`.

4.3.3 Construct. As described in the matching phase, the query formulates a set of bindings to AST nodes. These bindings are used during the constructing phase to map from the AST to a widget. In general, a binding is an access to the matched AST to construct a user interface and a closure that defines how to map back from the state of the user interface to the AST.

For example, we may bind the contents of a string to a user interface element, such as a color picker. During initialization, we access the string to find the color value, interpret it, and assign it as the picker's color. For updating, we read the picker's current color, serialize it as string, and update the AST. If a value remains read-only in the widget, the updating closure can be left empty.

For embedding a bound AST node in our widget, we replace the matched node with a placeholder in the original AST that references the node. We can then add the node

to our widget instead. In the update closure, we temporarily swap the placeholder and the original node to provide a complete AST again.

The above two examples of bindings concern a one-to-one mapping from one source to one destination. The system also supports sub-matches within containers. For example, if our replacement binds an array node, we can create a nested *repeat binding* that

- specifies a query to be run inside the array to find and bind existing children,
- a snippet that specifies a template for the sub-AST when the user inserts a new element into the array, and
- a function that maps each child to the desired user interface within the widget.

In the below snippet, we take the matched array binding, specify that we want to query for any children that it contains that are arrow functions, and specifically we extract their body. We provide a snippet for creating a new element in the array. And finally we provide a closure where each matched child is turned into a user interface.

```
self
  createRepeatBindingFor: '@array'
  childQuery: '(arrow_function body: (_) @root)'
  new: '() => _expr'
  build: [:bindings |
    self addMorph: (bindings at: '@root')]
```

4.3.4 Edit and Sync.

During construction, we establish a mapping between the AST and the user interface. Edit operations by the user are first only reflected in the user interface of the visual replacement. Whenever the AST is traversed, in particular for purposes of turning the AST back to text for saving textual languages, we run the update closures of all bindings. As such, from the point of view of an outside observer such as tools, the mapping is always up-to-date.

As embedded nodes are moved between their original place in the AST and the visual replacement's user interface between update calls, information that attaches to the node, is moved with it. So, if an error-checking tool attaches an indicator to an AST node, the indicator is also shown in the embedded node in the visual replacement. Other tools that rely on the context of an entire module, such as semantic autocompletion, continue to work in the same manner, as, from the perspective of the tool, even the embedded code is still inside the AST.

4.3.5 Runtime.

As described above, visual replacements do not exist as a separate concept from the point of view of external tools. These also include the compiler or runtime of the program containing visual replacements. Instead, they appear as regular code, sometimes adapted to serve specific needs of a replacement, such as a marker.

One particular advantage of the visual replacement mechanism is, however, its ability to handle instrumentation code. For example, to implement a watch that reports the value of a wrapped expression back to the programming environment, the expression can be wrapped in a marker such as `vrReport(expression=3 + 5, id=32123)`, where the `vrReport` function is an injected function that sends a serialized form of the evaluated result of the given expression, along with an `id` to uniquely identify that watch, to the programming environment, then returns the evaluated result.

The means to communicate between the runtime and programming environment depend on the runtime's constraints. In our implementation, runtimes that execute directly on top of the user's operating system use TCP connections. The programming environment hosting the visual replacement opens a TCP socket on a random port, invokes the runtime while passing the chosen port as an environment variable, and the injected code sends a JSON object to the waiting socket at the indicated port. For web applications, we use a HTTP request to an endpoint opened by the programming environment while we listen to the runtime's activity.

While the instrumented code is necessarily jarring to look at, visual replacements will hide all but the parts relevant to the user, which would typically leave a visualization of the runtime value and the expression to be evaluated, if it is to remain editable by the user. Unlike other means of instrumentation, this approach requires no modifications to the runtime. It is, however, also limited to work with the code where visual replacements have been used.

4.4 Language-agnostic Visual Replacements

While visual replacements, once created, only match for language constructs in the language its query is written for as described in subsubsection 4.3.2, adding support for further languages is simple. Two parts are necessary for a visual replacement to work with a given language.

First, the visual replacement's query needs to be adapted to the syntax of every new language. In the ideal case, differences between the languages in the ways bindings are extracted can be handled in the query directly. For example, in some languages the string node is terminal, in others may include multiple `string_content` nodes. If a more complicated extraction logic is required, authors can add instructions to manually perform a translation step to unify the binding's format, for example in the form of a pre- and post-processing step to the access and update part of a binding as described in subsubsection 4.3.3.

Second, users need to define a language runtime mapping that informs the programming environment how programs can be executed. In particular, the runtime mapping should define two mechanisms: an *evaluate-in-scope mechanism* that allows the environment to pass a source code string and an AST node, where the runtime mapping tries to invoke its runtime with the given code as close to the given node

as possible; and a *watch mechanism* that replaces a given expression with an expression that still returns the same value and importantly only evaluates it once, but also reports the value to the programming environment, as described in subsubsection 4.3.5.

Visual replacements that are written for a specific API, or aspects where semantics that concern the visual mapping differ between languages, a port to another language may not make sense or require more effort.

5 Case Studies

A number of practical considerations arise when putting the outlined visual replacement concept into practice. In the following, we describe case studies of diverse uses of visual replacements to illustrate these considerations.

5.1 Matplotlib

Data visualization is a crucial task in the field of data science [4]. Python is one of the primary languages used by data scientists [4]. In this context, matplotlib, developed by John Hunter in 2007, stands out as one of the oldest and most popular library for data visualization in the Python ecosystem [8].

Several approaches have been explored to enhance the integration of plotting libraries with programming environments. For example, *mage* facilitates binding code to corresponding custom user interface widgets in a Jupyter notebook extension [9]. The respective widget allows changing the type of the plot, switching between pre-defined parameters, and viewing the resulting plot immediately in the user interface. However, the authors acknowledge a reduced expressiveness of their widgets system compared to writing code directly. In *B2* [17], a two-way feedback binding mechanism enables interaction with the visualization to update the underlying code and vice versa. In this way, the gap between visual and textual programming is bridged, specifically in terms of layout, semantics, and temporal aspects; layout referring to the mismatch of traditional one-dimensional code layouts vs. multi-dimensional visualization layouts, semantics referring to the incompatibility of writing code and interacting with visualizations, and temporal aspects referring to the lack of persistence when interacting with visualizations compared to writing code.

As opposed to the other discussed approaches that require explicit bindings between code and user interface, our implementation of a bridge between visual and textual programming for data visualizations uses implicit visual replacements exclusively. Every encountered API call to matplotlib is automatically replaced with a visual replacement. Consequently, two challenges arise: first, how can we map large, relevant parts of the API surface with minimal effort, and two, how can visual replacements provide useful feedback to authors of visualizations across the varied functions of API calls of matplotlib?

For the first challenge, our approach leverages the extensive documentation provided by matplotlib to guide the effort to make the API more discoverable and provide better feedback. Within this context, a reusable layer of generic visual replacements on top of the fundamental functionality as described in section 4 emerged. For the most basic replacements, we extracted thumbnails for common plot types from already existing documentation[1] and displayed those adjacent to the code text to provide visual anchors in the otherwise uniform looking source code. The corresponding visual replacement makes no assumptions concerning programming language or API and can be reused in any place small illustrations are appropriate.

Similarly, a set of visual replacements mimicking widgets as they would appear in a form emerged, which replace literals and rewrite the source code when changed. For parameter values that are discrete and limited, such as typical enums, users select options via dropdown menus and may even benefit from an icon for each option, again extracted from documentation in matplotlib's case as shown in Figure 1a. Booleans become checkboxes, colors receive a color picker, or numbers can be turned to spinboxes or sliders.

Some parts of matplotlib's API we deemed both non-generalizable and difficult to understand without a domain-specific view. One such example is the `dashes` parameter of the `plot` function to create a dashed line. It sets the distances between lines and gaps through a tuple of values. To support user's understanding and provide immediate feedback on the outcome of their configuration, we show a dialog when the user begins to edit any value that previews the resulting dashed line. Extending that dialog with direct manipulation of line distances would also be feasible.

Better means for exploring the possibility space is a major advantage of graphical user interfaces, which both *mage* and *B2* incorporated, such that users can explore the parameter space by simply clicking on areas of interest. In our implementation, we made use of the sidebar known from block-based editors to demonstrate to users what options are at their disposal. This sidebar takes into account the cursor position, e.g., if the cursor is within the parentheses of the `plot` function, it brings the relevant parameters to the top, followed by general matplotlib functions.

Importantly, our design does not impose a ceiling in terms of the possibility space that users can interact with. Only parts of the matplotlib API are mapped to visual replacements but they are embedded within regular Python code. Consequently, users can always make use of the means of combination and abstraction offered by Python, or call functions that our implementation currently does not consider.

[1] https://matplotlib.org/cheatsheets (last accessed: July 2023)

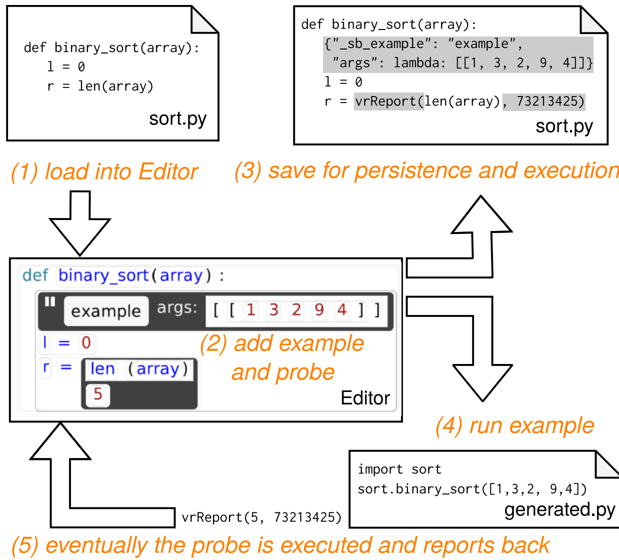


Figure 6. Use case scenario: (1) the user opens a Python file, (2) the user adds an example and a probe, (3) when saving, the visual replacements for the example and probe are written to the file and the editor automatically runs the example, (4) the editor generates a Python script that triggers the function that holds the example, (5) during the execution, the `vrReport` function is executed, which reports the result of the expression `5` back to the editor. To distinguish between values reported for different examples, the editor injects a different `PORT` environment variable for each run of an example that `vrReport` uses.

5.2 Babylonian Programming

Babylonian Programming [14] is an approach to example-based live programming. It adds examples to language constructs that provide entrypoints for execution closer to the relevant code than running the entire program. Users can add probes to show runtime values, aggregated by the example during whose execution the value was reported.

We describe here an implementation of its core elements, examples and probes, using the visual replacement system. An example in our implementation annotates a function and consists of three parts: a name, saved as a string, an embedded expression that returns an array of arguments to be passed to the annotated function, and an optional embedded expression that constructs an instance of an object, if the function is a method of a class.

For most languages, we store these three parts in a dictionary literal or, alternatively, an array. Both arguments and constructor are wrapped in a closure that is never evaluated as part of the main program. For Python, the serialized form in code of an example with the data it needs to store looks as follows:

```
# Example as stored source code
{
  "_sb_example": "example 1",
  "args": lambda: [2, 3],
  "self": lambda: MyObject()
}

# Query to match examples
(dictionary
  (pair
    key: (string) ="args"
    value: (lambda (_) @arguments))
  (pair
    key: (string) ="_sb_example"
    value: (string) @name))
```

The query looks for dictionary literals containing the magic `"_sb_example"` key to identify the explicit visual replacement. An equivalent query for JavaScript is almost identical, except for the closure and constructor syntax:

```
# Example as stored source code
{
  "_sb_example": "example 1",
  "args": () => [2, 3],
  "self": () => new MyObject()
}

# Query to match examples
(object
  (pair
    key: (string (string_fragment) ="args")
    value: (arrow_function (_) @arguments))
  (pair
    key: (string (string_fragment) ="_sb_example")
    value: (string (string_fragment) @name)))
```

The implementation of Babylonian Programming using visual replacements is illustrated in Figure 6. Probes are implemented using the watch mechanism described in subsection 4.4. Examples trigger execution whenever the open file is saved in the programming environment. The example will then evaluate a snippet of code made up of the invocation of the function call with the configured arguments and object. While the example is executing, all values reported by watches are collected and tagged for the executing example.

5.3 Regular Expressions

Regular expressions that exceed a certain level of complexity are often considered challenging to understand, especially by a person that was not its original author, due to its terse syntax. In response to that, tools such as regex101.com add syntax highlighting, live feedback, and explanations of syntax elements and their effect.

In this case study, we describe a "verbose mode" for regular expressions, where each part of an expression's syntax can be given a more or less terse appearance as seen in Figure 7. In addition, special no-op markers are added that allow users

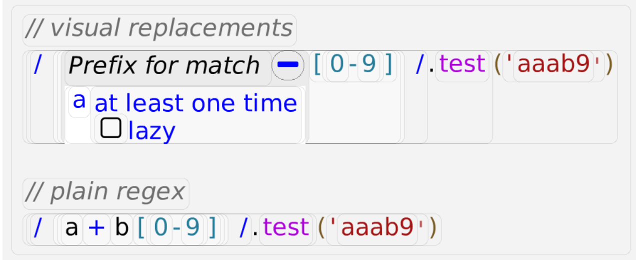


Figure 7. Verbose mode for regular expressions, embedded in JavaScript. Elements such as + are replaced with a verbose descriptor "at least once" and feature a checkbox to toggle their lazy mode. A collapsible comment annotates the first part of the regex.

to add comments to parts of a regex or mark it for reporting live-feedback.

Visual replacements help in another aspect: regular expressions typically exist as a language nested within another language. Through the replacement mechanism, a regular expression specified as a string in Python can first be replaced with a structured editor for the same regex. Then, each syntactic part of that regex can be replaced with its verbose pendant.

As most regular expressions do not have support for a comment syntax, our design offers two different solutions to persist information that is not supposed to be part of a match: information can either be placed next to the regular expression, indicating character offsets for the region they apply to, or embedded in a non-capturing group that should never appear, in the form of `(?:(?:annotation){0}(?:pattern))`. Here, the pattern is the part of the regex we would like to annotate and the annotation is base64-encoded arbitrary data that we want to store related to the pattern, for example a comment or a JSON object. While storing annotations in the encoded format compromises readability while in editors without visual replacement support, it also ensures that the regex remains self-contained and for example no fragile means of annotating, such as indices, are used.

6 Discussion

In this section, we discuss our approach of visual replacements concerning our requirements and their applicability in a cross-language context.

6.1 Requirements

In the following, we will consider the requirements established in section 3 and discuss each, considering our approach and insights gained from our case studies.

No new syntax. As evident from the design described in section 4, visual replacements match against the existing syntax tree of a language. Implicit visual replacements, by their definition do not require additional information. Explicit visual replacements make use of language specific means to annotate the AST with the means provided by the language. Notably, as demonstrated by the regular expressions case study, some languages do not offer any means to annotate the AST. Here, choosing a workaround that compromises readability or a standoff markup option, storing annotations next to the syntax tree may be the only options for authors.

The main advantage to be gained from not adding new syntax is to keep existing tool support working. While this does indeed work, linters and similar tools may report stored expressions and other generated code specific to a visual replacement as potential mistakes. While in an editor with support for visual replacements we can easily isolate and hide these warnings, programmers in other programming environments may perceive these as annoying when working in a code base with visual replacements.

Graceful degradation. Editors that do not support visual replacements should still offer a presentable representation to users. As seen in the Babylonian Programming case study, little boilerplate is added around the essential information: the expressions that make up the example. Dictionary keys replace labels in the user interface, making the association clear. However, again, some visual replacements may appear jarring or distracting when seen in source code, for example the pattern to denote an explicit slider `[3.14, "slider"][0]` used for some languages in our implementation. As such, we may have ensured that users outside an editor supporting visual replacements can understand what the visual replacement is supposed to mean and make an effort to preserve its structure during merging operations; however, they would tend not to look natural.

Avoid comments. While visual replacements can make use of comments, for the previously mentioned reasons of gaining some assurances in terms of syntactic structures from the host language, designers of visual replacements are encouraged and able to store information needed for their replacement as proper part of the syntax tree.

Static and dynamic information. Both static and dynamic information is available to visual replacements. Our case studies demonstrate helpful uses of both. We will discuss the complexity from their implementation below in subsection 6.2.

Open ceiling. As discussed in the matplotlib use case, which seeks to make a domain mapping that is expressed exclusively in code more accessible through visual elements, the elements only augment source code. Users can at any point choose to instead express logic using code, or mix visual replacements with code. The possibility to nest expressions within visual replacements allows designers of replacements great flexibility for their use case. Coupled with a watch, even a replacement on a dynamic expressions

rather than a literal, can provide a rich domain-specific display. Additionally, as visual replacements maintain a bijective mapping to source code, users can choose to work on source code instead of the DSR.

6.2 Cross-language Visual Replacement

In section 1, we described that visual replacements should work across different languages with minimal effort. The Babylonian Programming case study illustrates for a non-trivial use case to what extent this is possible.

Runtime: watch mechanism. In the platforms we targeted in our reference implementation (GDScript, Python, node.js with JavaScript and TypeScript, JavaScript on the web, Squeak/Smalltalk), the watch mechanism was comparatively simple to implement. Static languages that have very little run-time or compile-time reflection, such as C, are considerably more difficult targets, as general-purpose serialization as needed to report values back to the programming environment is not readily available within the language. Here, visual replacements may need to inform a watch about specific properties of the expression they are nesting, such that more sophisticated code generation can take place.

Runtime: execute context. In our reference implementation, executing an expression can be considered a best-effort, depending on the language. For Squeak/Smalltalk, where the runtime is continuously alive, even a highly specific context can be easily obtained. Contrarily, other languages require us to produce a file that imports relevant code or just run the entire program and can thus only approximate the effect of an isolated evaluation.

Matching the syntax tree. For a majority of visual replacements we have implemented, adding support for a new language did indeed only require adapting the matching query to the names of the language construct in that language. Common patterns that visual replacements use had to be rediscovered once per language, such as storing additional data in the regular expression case study, or storing expressions that should never be evaluated in closures.

6.3 Future Work and Conclusion

Visual replacements form a basic, cross-language, general-purpose mechanism for replacing source code with other representations while maintaining a tight integration with the syntax tree and language tools.

To better support cross-language visual replacements, future work could build another layer on top of the modified Tree-sitter queries described in subsubsection 4.3.2. Here, aspects such as string content, function call, or closure could be formulated as specific Tree-sitter query once and then reused in a compatible, generalized context, such that queries only have to be written once per language. A similar approach has been demonstrated by a Babylonian Programming implementation on top of GraalVM [11].

As of right now, invocation of visual replacements always occurs through statically available information. Instead, it could be possible to use dynamic information from the runtime to invoke visual replacements, for example to attach to AST nodes and explain type errors that occurred at runtime.

Finally, when further extending the use cases, we build up layers of reusable functionality on top of the visual replacement mechanism. The matplotlib case study shows how parts of its implementation, form widgets, can be designed in a way that is reusable across languages. In the context of the same case study, automatic derivation of visual replacements from documentation could pave the way for wide-spread availability of tightly integrated, rich inline documentation.

Acknowledgments

We gratefully acknowledge the financial support of HPI's Research School[2].

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] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 595, 16 pages. https://doi.org/10.1145/3544548.3580785

[3] Max Brunsfeld. 2020. Tree-sitter Queries. https://tree-sitter.github.io/tree-sitter/syntax-highlighting#queries [Online, accessed 09 July 2023].

[4] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's wrong with computational notebooks? Pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–12. https://dl.acm.org/doi/pdf/10.1145/3313831.3376729

[5] Patrick Dubroy. 2014. *Moonnchild.* Workshop on Future Programming (FP), SPLASH 2014. Retrieved 06 July 2023 from http://www.future-programming.org/2014/program.html

[6] Brian Harvey and Jens Mönig. 2015. Lambda in blocks languages: Lessons learned. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, USA, 35–38. https://doi.org/10.1109/BLOCKS.2015.7368997

[7] Marijn Haverbeke and CodeMirror team. 2007. *CodeMirror.* Retrieved 09 July 2023 from https://codemirror.net/

[8] John D Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in science & engineering* 9, 03 (2007), 90–95.

[9] 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

[2] https://hpi.de/en/research/research-school.html

[10] 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)*, Rebecca E. Grinter, Tom Rodden, Paul M. Aoki, Edward Cutrell, Robin Jeffries, and Gary M. Olson (Eds.). Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1124772.1124831

[11] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3426428.3426919

[12] 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

[13] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 859–869.

[14] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (feb 2019). https://doi.org/10.22152/programming-journal.org/2019/3/9

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

[16] Tamás Szabó, Markus Voelter, Bernd Kolb, Daniel Ratiu, and Bernhard Schaetz. 2014. Mbeddr: Extensible Languages for Embedded Software Development. *Ada Lett.* 34, 3 (Oct. 2014), 13–16. https://doi.org/10.1145/2692956.2663186

[17] Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging code and interactive visualization in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 152–165. https://doi.org/10.1145/3379337.3415851

Received 2023-07-17; accepted 2023-08-07