# Lorgnette: Creating Malleable Code Projections

Camille Gobert, Michel Beaudouin-Lafon

# LORGNETTE: Creating Malleable Code Projections

Camille Gobert
gobert@lisn.fr
Université Paris-Saclay, CNRS, Inria
Laboratoire Interdisciplinaire des Sciences du Numérique
91400 Orsay, France

Michel Beaudouin-Lafon
mbl@lisn.fr
Université Paris-Saclay, CNRS, Inria
Laboratoire Interdisciplinaire des Sciences du Numérique
91400 Orsay, France

## ABSTRACT

Projections of computer languages are tools that help users interact with representations that better fit their needs than plain text. We collected 62 projections from the literature and from a design workshop and found that 60% of them can be implemented using a table, a graph or a form. However, projections are often hard-coded for specific languages and situations, and in most cases only the developers of a code editor can create or adapt projections, leaving no room for appropriation by their users. We introduce LORGNETTE, a new framework for letting programmers augment their code editor with projections. We demonstrate five examples that use LORGNETTE to create projections that can be reused in new contexts. We discuss how this approach could help democratise projections and conclude with future work.

## CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; • **Software and its engineering** → **Software notations and tools**.

## KEYWORDS

Lorgnette, Projection, Semantic interaction

**ACM Reference Format:**
Camille Gobert and Michel Beaudouin-Lafon. 2023. LORGNETTE: Creating Malleable Code Projections . In *The 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23), October 29-November 1, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3586183.3606817

## 1 INTRODUCTION

While programming systems [40] exist in many shapes, languages have long been and remain one of the most prevalent tools to program computers [3]. According to recent reports from GitHub[1] and StackOverflow,[2] software developers commonly use many different programming languages. Some, such as Python and Java, have been ranked among the most used languages for years. Knowing a programming language is becoming a staple skill on the job market, taught to children and university students alike.

Inspired by the syntactic/semantic distinction common to computer science and linguistics,[3] we distinguish two paradigms for interacting with programming languages based on the class of conceptual objects they expose: *syntactic interaction*, in which users interact with the concepts *of* a language (such as its symbols), and *semantic interaction*, in which users interact with the concepts expressed *in* a language. As an example, semantic interaction allows to interact with a structure containing three numbers either as a colour or a 3D position depending on their meaning, whereas syntactic interaction only allows to manipulate such a structure as a generic language construct. While some syntactic techniques have recently proven successful, e.g., block environments such as Scratch [66], the genericity of the representations permitted by syntactic interaction can hardly bridge the gulfs of evaluation (interpreting code) and execution (modifying code) identified by Norman [53]. Semantic interaction techniques address this limitation by letting users manipulate representations that are closer to their mental models of the code than the syntactic form, thereby improving the *closeness of mapping* of the notation [32].

In this work, we focus on a subclass of semantic interaction techniques we call *projections*—a term adapted from projectional editing [75]. Projections are alternative interactive representations of specific fragments of code that complement syntactic interaction techniques such as text editing by letting users visualise and/or manipulate concepts expressed with the help of the language, such as a colour picker for interacting with a piece of code containing three numbers that represent a colour. Unlike fully visual languages and "no-code" systems, projections are compatible with (existing) general-purpose programming languages as they can target fragments of code with specific meanings, without requiring to represent the rest of the language any differently. Systems that support projections already exist, but they are often restricted to a single language and can only project macros or predefined code patterns, without letting programmers appropriate their programming systems [19]. They do not fully answer recent calls for more malleable programming systems, which should let programmers "*define new representations, according to their needs*" [49], embrace "*heterogeneity in programming practice*" [3] and integrate "*domain-specific interactive programming tools into general-purpose programming environments*" [39].

To address these limitations, we introduce LORGNETTE (pronounced "lor–niet"), a new framework that lets users create and modify the projections they want to use in their code editor. Unlike existing approaches, LORGNETTE targets both novice programmers

[1] https://octoverse.github.com/
[2] https://survey.stackoverflow.co/2022/

[3] This distinction was previously applied to programming by Shneiderman and Mayer [73], but theirs concerns *knowledge* whereas ours concerns *interaction*. Our definition of *semantic* is also quite general as we make it sensitive to the context, i.e., what Connolly and Cooke distinguish as *pragmatics* [15].

*UIST '23, October 29-November 1, 2023, San Francisco, CA, USA*

looking to use and tweak projections created by others; expert programmers willing to create projections, either by reusing an existing user interface or by creating one from scratch; and researchers interested in a platform for testing and evaluating projections for diverse languages. We first review related work. We then analyse 62 projections we collected by reviewing the literature and organising a design workshop with nine programmers. We present the concepts and implementation of LORGNETTE and compare it to similar systems. We evaluate LORGNETTE's capabilities by explaining how we created projections to help with five different situations. We conclude with the limitations of our approach and present directions for future work.

## 2 RELATED WORK

We review related work to situate our approach for authoring projections amongst the various programming practices, interactive systems and engineering techniques that exist.

### 2.1 Programming practices

Brooks [12] observed that a part of the burden of software engineering comes from the *accidental* complexity of the tools we use for programming. He argued that, unlike the *essential* complexity of creating software, this accidental complexity could be avoided. In reaction, Simonyi predicted "*the death of computer languages*" in 1995 [74]. In his view, users of programming systems should rather focus on the *intent* of a program—an approach he named *intentional programming* [75]. This vision led to software engineering paradigms such as *model-driven engineering* [71] and *language workbenches* [23], which require a certain upfront investment to define a model or a domain-specific language before it can be (1) edited by projecting parts of it as arbitrary representations such as text, formulas, diagrams, etc, and (2) transformed into executable code (or into a language that can be executed).

Almost 30 years later, though, computer languages are still alive and well. Old languages keep being used [41] and new languages keep being created, including for domains with WYSIWYG alternatives such as document authoring [14] and data visualisation [69]. Furthermore, alternative views have identified and argued in favour of less rigid programming practices. Live [77], exploratory [64] and *Babylonian-style* [63] programming allow and encourage building programs by making small, incremental changes and seeing the result in real time. Postmodern programming [52] rejects any "grand narrative" about a single best style of programming and encourages eclectic approaches instead, in which "messy is good" and "programs can exhibit faults in construction". Viewing programming as craft [10], in opposition to Dijkstra's vision [17], supports the idea that programming can also be an embodied skill rather than (just) a scientifically validated method.

These views not only recognise that requirements, environments and languages keep changing—making them *permanently beta* [51]—but also that programming systems should help programmers accommodate these changes. This echoes Dix's call for designing for appropriation [19], which encourages designers to create artefacts that can be used in unplanned and unexpected ways by their users. It is further supported by recent findings on the ability of users to use digital tools in unexpected ways by reasoning on the properties of their target [65] and examples of flexible tools for textual document authoring such as Textlets [34] and Potluck.[4]

Modern programming practices call for more malleable programming tools that work with diverse languages. Yet, existing systems with projections are not aligned with this goal: at best, users can create projections for one language only; at worst, they must do with what the developers of the code editor they use decided for them. Our approach seeks to help users appropriate their code editors by letting them adapt and create projections for any language without requiring experts to intervene.

### 2.2 Interaction with computer languages

The first and most popular technique to interact with programming languages is text editing, whose prevalence is directly inherited from the massive use of typewriters to program computers, which actually inhibited alternative notations that were common at the time of the first computers [3]. Early alternatives include structured editors such as the Cornell Program Synthesizer [78], which only allow programmers to make syntactically valid changes to a program, as well as direct manipulation of language primitives as in Boxer [18]. They have been followed by more modern approaches such as live data structures [20, 35] and block programming [7]. Artificial intelligence techniques have also been proposed to synthesise code using examples [25], sketches [21] and natural language [38]. However, they only help transform a different medium into code, without helping their users interact with already written code.

While some of these techniques have been advocated as "better" ways to program than text editing, the past few years have rather seen the emergence of *hybrid* programming systems that combine different interaction techniques, as illustrated by Droplet[5] and Pencil Code[6] (text and blocks), Nodes[7] and Enso[8] (text and graphs), and Stride [44] and Deuce [37] (text and direct manipulation). The *output-directed programming* paradigm yields other forms of hybrid programming systens, in which the alternative representation is the output generated by the code, which can be directly manipulated, as illustrated by systems such as Transmorphic [72] and Sketch-n-Sketch [36]. However, as Hempel et al.'s put it, "*manipulation of the final output alone will be insufficient*" and "*the intermediate process should be exposed [...] for manipulation*" [36, p. 5].

This restriction does not apply to projections, which use their own representations. They have already been applied to various contexts and languages, such as analysing data in Python [42, 82], inserting colours and regular expressions in Java [57], optimising loop parallelisation in C [85] and writing documents in LaTeX [30]. However, while projections have already proven successful, they have primarily been developed as isolated prototypes and ad-hoc tools for specific languages, forcing users to change their workflow to adopt them and hindering their integration into popular code editors. Our approach addresses these limitations by allowing users to mix and match the alternative representations that meet their

[4] https://www.inkandswitch.com/potluck/
[5] https://droplet-editor.github.io/
[6] https://pencilcode.net/
[7] https://nodes.io/
[8] https://enso.org/

needs, in the same spirit as complementary syntactic interaction techniques in hybrid code editors.

## 2.3 Alternative notation engineering

Macros have traditionally been used to extend a textual language's notations by inserting parameterised text in the code or nodes in the syntax tree. Since then, other mechanisms for defining custom syntactic sugar [22] and literal notations [55] that can be packaged into libraries have been proposed. However, all these approaches only support new textual notations.

To support alternative visual notations, code editors can be augmented either by their developers or by their users. On the one hand, developers can hardcode alternative visual notations in specialised code editors. This is the approach taken by language workbenches such as Barista [43] and JetBrains MPS [80], which allow to define a language, program how to represent each syntax tree node, and generate a custom program editor for this language. This approach is also used by visual editors that use their own internal representations and allow to import/export code written in other languages, such as Envision [5] for Java/C++ and LyX[9] for LaTeX. On the other hand, some systems let their users free to decide how to represent language elements on the fly. This is typically possible in image-based programming systems such as Smalltalk [31], Pharo [9], Lively[10] and the Glamorous Toolkit,[11] in which data, code and its output are all in-memory objects that can be inspected and modified by any tool that can read them. It can also be done using visual macros written in the same language than the one they can be used in. This approach has recently been demonstrated for Racket [2] and Hazel [56]. It requires an editor capable of interpreting the code of the macro at edit-time, which must specify (1) how to build the user interface of the macro and (2) how to generate code from the model of the user interface.

Independently of the approach, keeping multiple representations synchronised is challenging. This problem was first studied as the *view-update* problem in the database community. It later inspired Foster et al.'s *lenses* [26], which reify bidirectional transformations between two data structures that can be composed. To facilitate the task of updating code in output-directed programming systems, special languages with semantics that preserve provenance information [13] have recently been developed and applied to HTML editing [48], SVG image manipulation [36] and data visualisation brushing [59]. However, we are not aware of any application of this technique to create alternative notations for local fragments of code.

Our approach to creating malleable projections is inspired by systems with user-defined visual notations, but we want to go beyond working with a single language or in a siloed image-based environment. We seek to apply the concept of lenses to code editors so that users can link fragments of code written in any computer language with data structures that can then be used as models for user interfaces that help interact with these code fragments.

[9] https://lyx.org/
[10] https://lively-next.org/
[11] https://gtoolkit.com/

# 3 STUDY OF PROJECTION DESIGNS

To further study what projections are used for and what our framework should prioritise, we analyse projections that we collected by organising a design workshop with programmers and reviewing projections published in the literature. We present our methodology for collecting projections and report on the results.

## 3.1 Design workshop

Our first source of projections is a participatory design workshop that we organised with programmers who were asked to imagine and design projections they would like to use.

*3.1.1 Participants.* We recruited 9 participants (6 men, 2 women, 1 prefer not to say; age 18–44) by posting messages on the mailing list of a computer science department. 2 participants were research engineers, 3 were Ph.D. students, and 4 were associate professors or researchers in computer science, human-computer interaction and information visualisation. All participants had at least 5 years of experience with programming, and 6 (66%) had more than 10 years of experience. All of them program at least once a month, and 7 participants (78%) do it at least once a week. In addition, all were familiar with multiple programming languages. Participants did not receive any compensation for their participation.

*3.1.2 Setup.* The workshop was organised in person and lasted about 2 hours. Participants were put in groups of 3 so as to maximise the diversity of the profiles in each group (based on data collected in the questionnaire). All the tasks were on paper. Participants were provided with all the drawing material they needed.

*3.1.3 Procedure.* Before the workshop, we asked participants to fill in an online questionnaire to collect demographics and information about their experience with programming. To prompt participants, we also asked them to write about a situation in which they wished they had access to an alternative representation of a piece of code. At the start of the workshop, we asked each participant to summarise the situation they wrote about .

We then introduced the concept of projection and presented the three tasks to perform. We showed videos of projections that we created demonstrating code manipulation using a colour picker, a 2D table and a style inspector. We also briefly explained how projections work, insisting on the fact that the same user interface can be reused in different contexts.

In the first task, we asked participants to list as many ideas of projections as they could think of—first by proposing three "good" ideas and two "bad" ideas individually, and then by discussing ideas within their group. Each idea had to include a title, a short description of the situation it applies to, a short description of the user interface of the projection, as well as one or several goals among: discovering features, understanding code, modifying code, debugging, explaining, or any freely specified goal. In the second task, we asked each participant to choose two ideas generated by their group and to design the projections they describe. Each design had to include a description of the context in which the projection could be used, an example of the code or data to represent, and an annotated sketch of the user interface of the projection. In the third task, we gave each group the designs created by another group and asked participants to choose one design and to apply it to a

new situation. Each redesign had to include a description of the new context the projection could be used in and an example of the new code or data to represent. They could optionally include a new version of the sketch if the user interface had to be adapted to work in the new situation.

*3.1.4 Data collection.* We collected and transcribed all the documents produced by the participants. The full list of designs and redesigns created by the participants, as well as examples of what they produced, is available in Appendix A.

We ignored 3 designs and redesigns that we categorise as generic code editor features rather than projections: making type-aware suggestions (D5), performing global renaming (R6) and displaying the definition of a symbol (R8). We did not treat D15 (highlighting polyglot code) as such as it may not be supported by a code editor if it requires user knowledge, e.g., to know that certain strings are written in a special language that can be parsed.

## 3.2 Examples from the literature

To increase the diversity of the projections collected during the workshop, we also collected projections from systems published in the literature. We specifically considered projections, and therefore excluded syntactic interaction techniques as well as other kind of code augmentation techniques [76].

Since the term *projection* is not commonly used to describe the kind of interactive system we are interested in, we did not use a systematic approach such as a keyword search. Instead, we (1) reviewed systems presented in articles published in the past few years to major conferences/journals in the fields of both human-computer interaction and programming systems[12] and (2) reviewed references of potential interest in the articles we selected in the first step. We only retained projections that were part of working systems. We therefore ignored unimplemented ideas of potentially useful projections, such as interactive chemical formulas [30] and table-shaped diagrams for TCP messages [2]. This method should not be considered as an extensive or a systematic review of the literature on projections, and we do not claim that the results fully capture the variety of projections available in published systems.

## 3.3 Results

The participants produced 44 projection ideas in the first part of the workshop, of which 23 were developed into actual projection designs, and we collected 39 from the literature, resulting in a total of 62 projections. We analysed the collected projections both in terms of contexts of use (which situation are they used in?), data (what is projected?) and representation (how is it projected?). We list the projections we reviewed and summarise our findings in Table 1, and we report the full lists of designs created during the workshop and articles we collected in Appendices A and B.

*3.3.1 Contexts of use.* Projections can target different audiences such as software developers, domain experts, e.g., to create electronic circuit [79], teachers, e.g., to create assignments (D14, D16, R2, R3), and students, e.g., to learn Rust [1]. They can also serve very diverse goals: among the 44 projection ideas that were proposed by workshop participants, 26 (59%) were designed to help them understand code, 24 (55%) to modify it, 16 (32%) to debug it, and 12 (27%) to explain it or to discover new features. One participant was also interested in using a projection to generate synthetic data (D12).

Projections can be used in diverse types of programming systems. In the literature, they are used in regular code editors (27/39), notebooks (5/39) and specialised environments (7/39). In the workshop, this was often unspecified, but most projections seem to be designed for code editors and two for terminals (D4, D9). They can also work with multiple languages—not just programming languages—including Python, Java, C, HTML, CSS, JavaScript, Swift, Rust, Racket, Haskell, Hazel, LaTeX, Markdown, YAML and others.

*3.3.2 Data.* Projections can rely on both *static* data available at all times, e.g., the code of a LaTeX table, and *dynamic* data available only during execution, e.g., the content of a Python dataframe. Some projections use both: for instance, mage's image editor [42, fig. 3] gets the image to edit from the memory at runtime but reads the area to crop in the code. Furthermore, one workshop participant proposed a projection idea (that was not turned into a design) to visualise the introduction of a merge conflict with another branch of their versioning system, which suggests that some projections may need *environmental* data that is not directly related to the code or its execution, such as data from the version control system.

When a projection represents code, the scale of the code also differs from one projection to another. Most projections only represent local fragment of code (50/62), e.g., a colour value or a local tree transformation. Others either represent entire files (6/62), e.g., a list of timestamped 3D positions or a code assignment, or code located in multiples files (6/62), e.g., the structure of a container in a HTML file and the related style in a CSS file. This echoes the multiple scales of code patterns that have been identified in the literature: beacons [81] and nano-patterns [29] at the instruction/line level; micro-patterns [28] at the class/file level; and design patterns [27] at the multi-class/multi-file level.

*3.3.3 Representations.* More than half the projections we reviewed (37/62) take the form of either a grid, a graph or a form, which seem to be very versatile user interfaces for projections. The rest of the projections use various user interfaces, including some highly specific ones that were used in only one situation, such as maps (D12) and music staves (D1). While the same user interface can be reused to represent different concepts, some concepts can also be represented by different user interfaces, e.g., a state machine can be represented both by a grid and a graph. Regarding interactivity, about three user interfaces in four allow to modify the code interactively (45/62), while others serve as static alternative representations of the projected data.

Projections can also be displayed at different locations, regardless of their user interface. In the literature, this includes *inline* projections (11/39), which mix with textual code; *side* projections (14/39), which appear next to the code; *detached* projections (10/39), which appear in a different panel; and *embedded* projections (4/39), which appear in another document. Besides a few projections created with JetBrains MPS, none of the systems we reviewed let users change the display location of a projection—even though that may benefit some user groups, as discussed by Omar et al. [56, §5.3].

[12] This notably includes ACM CHI, ACM UIST, ACM SPLASH, ACM PLDI, IEEE VL/HCC and the Programming journal.

**Table 1: List of projections we reviewed. Source represents the workshop design numbers as specified in Table 3 or references to published systems as specified in Table 4 (see Appendices A and B).**

| Representation | Projected concept | Source |
|---|---|---|
| Form | Colours | [56, 57] |
| | GUI component properties | [58] |
| | Graphical properties | [58] |
| | Animation properties | [58] |
| | Form building | [2] |
| | Regular expression | [57] |
| | Configuration file | D3, R5 |
| | FFmpeg configuration | D4 |
| | Pandoc configuration | D10 |
| | Documentation comment | D6 |
| | Shell expansion | D9 |
| | Code assignment | D14, D16, R2, R3 |
| Graph | List | [24, 35] |
| | Tree | [2, 24, 62] |
| | State machine | [24, 79] |
| | Rust's ownership | [1] |
| | Runtime stack and heap | [33] |
| | Reactive stream | [16] |
| | Call graph | [45], D7 |
| Grid | Dataframe | [42, 56, 61] |
| | State machine | [79] |
| | Document table | [30] |
| | Grid layout | [30], D11, R9 |
| 2D plot | Dataframe | [42, 82] |
| | Machine learning metrics | [83] |
| | Loop parallelisation | [85] |
| 3D plot | 3D trajectory | D8 |
| | 3D object | R7 |
| | Transform reference point | R1 |
| Typeset mathematics | Coded formula | [30, 43, 62], R4 |
| | Optimisation problem | D2 |
| 2D game board | Conway's *Game of Life* | [62] |
| | Tsuro | [2] |
| Circuit diagram | Hardware circuit | [47] |
| | Quantum circuit | [4] |
| Image editor | Image transformation | [30, 42, 56], D13 |
| Marble diagram | Reactive stream | [6] |
| Music staff | LilyPond score | D1 |
| Map | Geodata synthesis | D12 |
| Sequence diagram | Variable access | D17 |
| Highlighted code | Polyglot code | D15 |

## 3.4 Discussion

The results of the study depict a very eclectic design space for projections. Projections can be used in different sorts of programming environments and languages by both experts and non-experts. They rely on very different types of data that may originate from the code, its execution or the environment it is written in, and they can be represented by very generic or very specific user interfaces. A single user interface can be applied in different situations, and a single concept can benefit from multiple user interfaces.

Yet, existing program editors with projections are built using paradigms that only support certain subsets of that design space. For instance, projectional editors generated by model-driven engineering systems and language workbenches only support alternative representations for predefined syntax tree nodes in a single language, which cannot exploit runtime data since the language must be transformed before it can be executed. They also make the cost of creating projections very high since it requires having access to the sources of the editor, creating them from scratch (at best, parts of the implementation could be copied from other projections), and recompiling the model into a new projectional editor. Image-based programming systems are more appropriate for exploiting runtime data since they blur the distinction between code and data, but like projectional editors, they require to use their own language and do everything in their isolated ecosystem. Visual macro systems make it easier to create projections since they can do it from within the language to project itself, but this hardly promotes reusing the same user interface across languages, even though concepts common to multiple languages may benefit from the same kind of projection.

Overall, all these approaches still remain specific to a single language. They are often restricted to projecting specific AST nodes and do not support targeting arbitrary code patterns. Furthermore, they do not lower the cost of reusing generic user interfaces, even though we observed that three are used in 60% of the projections we collected.

# 4 THE LORGNETTE FRAMEWORK

In light of these findings, we introduce LORGNETTE, a new framework that allows users to create and modify the projections available in their code editor. LORGNETTE targets multiple categories of users, from novices who only want to tweak existing projections to seasoned programmers willing to create projections with custom user interfaces to accompany a library they work on. We present the concepts of LORGNETTE, describe its implementation and compare its features and properties to those of related systems.

## 4.1 Concepts

LORGNETTE is a framework for augmenting code editors with projections. It is not a code editor in itself, but a means to instrument a code editor so that its users can freely create, modify and delete projections by themselves without needing the code editor developers to intervene. To be compatible with LORGNETTE, a code editor must expose an **environment** that gives LORGNETTE access to resources (such as files and debuggers) and a **view** in which LORGNETTE can display projections and process user events. Once LORGNETTE has been configured for a code editor, users can start augmenting it with projections by writing projection specifications. A **specification** is a blueprint for a projection that tells LORGNETTE when and how to create it. It has three main responsibilities: extracting the appropriate resources, mapping them to a model, and pairing it with a user interface. The whole process of turning a specification

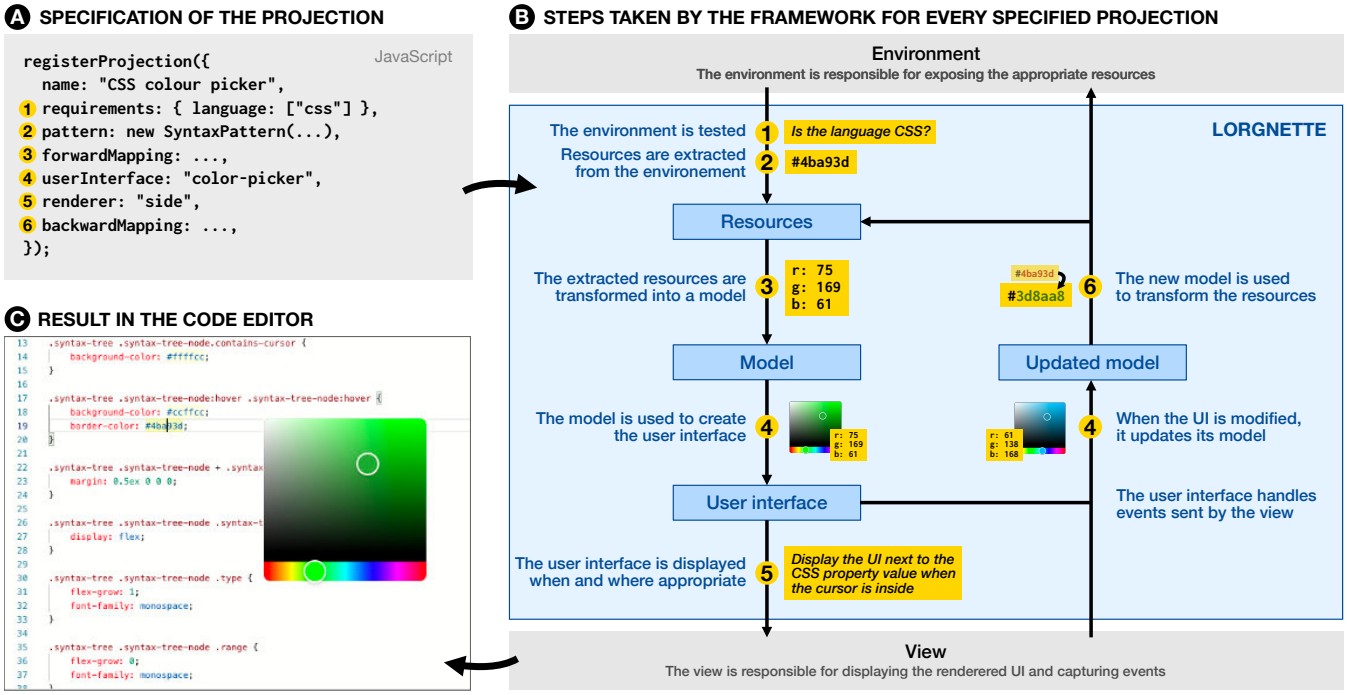



**Figure 1: Description of the process followed by LORGNETTE to instrument a code editor with a colour picker for hexadecimal colour codes in CSS. A The JavaScript specification of the projection. B The process followed by the framework to turn the specification into actual projections. The yellow labels show examples of what is created or performed at each numbered step. C The resulting augmented code editor. The displayed colour picker represents the colour coded in hexadecimal at the cursor's position (line 19).**

into a projection is depicted in Figure 1. The numbers in yellow discs in the text below refer to the six steps shown in the Figure.

*Extracting resources.* The first responsibility of a specification is to list its **requirements** (1), i.e., the conditions that must be fulfilled for the projection to be created. They typically represent assertions about the environment, such as the language(s) the projection is designed for. Moreover, they must also specify what are the **resources** (2) needed by the projection, i.e., to create a model for the user interface. Resources represent different sorts of data that can be used by projections. The main type of resource is the **fragment** of code that is being projected, either in the form of a range in the text (a textual fragment) or in the form of a node in the syntax tree if the language can be parsed (a syntactic fragment). Specifications must include a **pattern** that will be used by LORGNETTE to search for matching fragments in the code, each of which will be projected. These patterns can either be textual, e.g., using a regular expression, or syntactic, e.g., using an assertion that must be true for syntax tree nodes that can benefit from the projection. If the environment supports it, resources can also include runtime information, such as the value of a variable at a certain point in time. This requires to write **runtime requests** in the specification, which specify when and how LORGNETTE should query the runtime (such as a debugger running the code that is being projected) and whose responses will be provided to the projection as resources. LORGNETTE currently supports these three types of resources (textual fragments, syntactic fragments and runtime information), but other types of resources could be made available to projections as well, such as local files or environment variables.

*Mapping resources to a model.* When all the requirements to create a projection are met, the requested resources are extracted and passed to the **forward mapping** (3). The forward mapping is an arbitrary function that is responsible for processing the resources in order to return a valid model $M$, such as by extracting useful information from the code fragment. Whenever the state is modified, typically through the user interface, the new model $M'$ is passed to the **backward mapping** (6), possibly along with extra data (such as the previous model and other information on what changed). The backward mapping is an arbitrary function that is responsible for updating the underlying resources, e.g., by modifying the code fragment, so that applying the forward mapping would produce $M'$, i.e., so that the set of forward and backward mappings form a *well-behaved lens* [26]. In case the state is never modified, such as when using a projection to display a static representation, the backward mapping can be left undefined.

*Rendering the user interface.* Once the model has been generated by the forward mapping, the **user interface** (4) can be instantiated using $M$ and provided to the **renderer** (5). The user interface has the responsibility of deciding *how* to represent the data, process events such as clicks and key presses, and update the model when

appropriate; while the renderer has the responsibility of deciding *when* and *where* the user interface should be displayed in the code editor. For instance, by switching between two different renderers, the same user interface could be displayed either in a separate panel or next to the cursor when it is inside the code being visualised.

*Reusing patterns and mappings as templates.* While requiring users to write arbitrary mappings between resources and a model allows LORGNETTE to support very diverse situations, writing bidirectional mappings from scratch is effortful and hard to reuse across similar situations. To avoid this complexity, other systems such as Codelets [58] and mage [42] only allow to specify text templates with "slots". The slots can be written when the user interacts with the projection and read again when the code is directly modified. This solution has the advantage of being easy to use, but it is too simple to support a number of common situations. For example, it does not support creating slots in an unsorted key-value list—such as objects in Python or JavaScript—since all the possible orders would have to be enumerated.

To help users create projections for this kind of common situations, LORGNETTE includes its own type of templates. In LORGNETTE, a **template** reifies a procedure for creating a model containing key-value pairs. Each template specifies slots, which are not restricted to predefined ranges and whose meaning is completely template-specific. Each slot must have a **key**, which identifies the slot's value in the model, and an **evaluator**, which tells the template how to turn the text content of the slot into a usable value. A slot specification may also contain other information, such as a default value that should result in the slot's deletion in the code, e.g., to avoid cluttering a configuration object with default options. For example, a template for named arguments in Python lets users specify which function names they want to target, which arguments they are interested in (keys) and whether to parse each value into a string, a number, a boolean, etc. (evaluators). This template automatically generates resource specifications and mappings that result in an model where the keys are argument names and the values are evaluated argument values. Users are not restricted to use the model as is: templates can specify arbitrary functions to further transform the model into the shape expected by the user interface.

## 4.2 Implementation

LORGNETTE is implemented as a library written in TypeScript with React.[13] We open-sourced the code at https://github.com/exsitu-projects/lorgnette. React allows to exploit the large ecosystem of React components to quickly experiment with new user interfaces for projections, but it is not a core requirement of the implementation. In its current version, LORGNETTE can parse code written in five languages (JSON, CSS, Markdown, JavaScript/TypeScript and a subset of Python). It also includes six user interfaces (a colour picker, a table, a form, a file tree, a 2D plot and a regular expression diagram), three renderers (next to the code, in a popover and in a popup) and four templates (named groups in regular expressions, JSON objects, JavaScript objects and arguments of Python function calls). At the moment, supporting new languages and adding new user interfaces, renderers and templates requires to modify the source code of the framework. However, we plan to create a plugin architecture so that users can extend the framework without having to edit the framework's code.

We used LORGNETTE to create two different code editing environments: a playground in the form of a webpage, and a custom editor for the Visual Studio Code[14] (VSC) editor. The playground includes a number of examples to try projections for different situations and in different languages. When the language can be parsed, the syntax tree of the document can also be displayed next to the code editor, a convenience for writing syntactic patterns. The custom VSC editor is a custom extension for VSC that replaces the standard code editor of VSC by a similar-looking code editor that can be augmented with projections. It consists of two parts: a webview that runs an instance of the Monaco code editor instrumented with LORGNETTE, and a core that gives the webview access to some of VS Code's extension APIs via message passing.

One of the major differences between the two environments is that the custom VSC editor supports runtime queries. To do so, the extension converts runtime queries emitted by LORGNETTE in the webview into messages sent to debuggers supported by VSC using the Debug Adapter Protocol (DAP).[15] For each runtime query, the core sets a breakpoint at the start of the code fragment of the projection that emitted the query. When the breakpoint is hit by the debugger, the core (1) retrieves the current thread and frame IDs, (2) uses them to ask the debugger to evaluate the runtime query's expression using an `evaluate` request, and (3) resumes the execution. When the debugger answers the request, the extension captures the response, pairs it with the corresponding query, and forwards it to the webview, so that LORGNETTE can update the projection that made this query. Since the DAP is designed to be language-agnostic, this mechanism works with multiple languages and debuggers without requiring any change to the code (besides writing runtime queries in the appropriate language). In our tests, we were able to successfully use runtime queries with VSC's JavaScript debugger, Google Chrome's JavaScript debugger and VSC's Python debugger.

## 4.3 Comparison with existing systems

To explain what makes LORGNETTE unique, we compare it to eight other systems that also allow to create projections for code editing environments. We focus on the type of resources that can be projected, as well as on five properties of the resulting projections: *persistence*, *bidirectionality* and *composability*, which have been previously identified by Omar et al. [56] and Gobert and Beaudouin-Lafon [30], and *malleability* and *language agnosticism*, which we identified and named for the purpose of this work. The systems and the results of the comparison are summarised in Table 2.

*4.3.1 Type of resources.* We compared the type of resources that can be used to create projections: the text of the code itself, the nodes of the syntax tree that models it, as well as information only available at runtime. To the best of our knowledge, LORGNETTE is the only framework that supports all three. Barista, Envision and MPS only support syntactic patterns, and mage only supports textual patterns. The other systems are even more restricted, as

[13] https://react.dev/

[14] https://code.visualstudio.com/

[15] https://microsoft.github.io/debug-adapter-protocol/

**Table 2: Comparison of LORGNETTE with eight other systems that explicitly support projections in code editing environments. Orange values represent partially positive answers, and dashes represent negative answers. The meaning of the columns is explained in subsection 4.3.**

| System | Supported resources | | | Persistent | Bidirectional | Composable | Malleable | Language-agnostic |
|---|---|---|---|---|---|---|---|---|
| | Text | Syntax | Runtime | | | | | |
| Barista [43] | – | Yes | – | Yes | Yes | – | – | – |
| Envision [5] | – | Yes | – | Yes | Yes | – | – | – |
| MPS [80] | – | Yes | – | Yes | Yes | Yes | – | – |
| Graphite [57] | – | Constructors | – | – | – | – | – | – |
| Codelets [58] | Snippets | – | – | – | Yes | – | – | Yes |
| Visual syntax [2] | Macros | – | – | Yes | Yes | – | Yes | – |
| mage [42] | Yes | – | Yes | Partially | Yes | – | – | – |
| Livelits [56] | Macros | – | Yes | Yes | Yes | Yes | Yes | – |
| LORGNETTE | Yes | Yes | Yes | Yes | Yes | – | Yes | Yes |

they can only provide projections for class constructors (Graphite), predefined code snippets (Codelets) or user-defined macros (Visual syntax, Livelits), meaning that projections cannot be made available for existing code. In addition, only mage and Livelits support projections that use information from the execution of the code.

*4.3.2 Persistence.* Persistent projections remains available after they have been used once. LORGNETTE, like most other systems, offers persistent projections. Graphite and Codelets' projections do not persist as they are only designed to help users configure pieces of code when they are inserted, and mage's only partially persist as they require to re-evaluate code cells to turn the magic commands into projections again.

*4.3.3 Bidirectionality.* With bidirectional projections, the code can be updated by interacting with the projection and conversely. All the systems we reviewed but one are bidirectional, and so is LORGNETTE. The only exception is Graphite, which only allows to specify the code to generate by interacting with the user interface of its projections. This is a similar limitation to many code generators that help generating code once but cannot interpret edits made to the generated code, as in online tools that help write tables[16] and configure CSS properties.[17]

*4.3.4 Composability.* Composable projections can include other projections, e.g. by projecting the code of a table cell in a projection that arranges the code into a table. Only Livelits and MPS have this property. At the moment, LORGNETTE does not support composable projections, as a projection can only be created from the content of a code editor, not from the content of another projection.

*4.3.5 Malleability.* A malleable system lets users create, modify and delete projections. LORGNETTE's paradigm is specifically designed to support malleability by separating the specifications of the projections from the code editor. Visual macros for Racket and Livelits also achieve this property, but they are only meant to be used in a single language and do not offer as much control on the projection as LORGNETTE (such as when and where to display it). Other systems could be argued to be malleable by supporting extensions, but it typically requires a much higher amount of work, such as creating a custom code editor panel with its own projections in VS Code or completely specifying the syntax, type system, etc. of the language to project in MPS.

*4.3.6 Language agnosticism.* A system is language-agnostic when the user interface of a projection can be seamlessly reused across different languages. This must be distinguished from *polyglotism*, which qualifies a system that can distinguish among languages embedded in other languages. The only two systems with this property are Codelets and LORGNETTE. Codelets supports code snippets written in any language as it only treats them as text. In addition to supporting textual resources, LORGNETTE also supports cross-language syntactic and runtime resources by abstracting over language-specific features with concepts such as syntactic patterns (which work the same with any syntax tree) and runtime requests (which work the same with any DAP-compatible runtime).

Overall, LORGNETTE is the only framework that supports all three kinds of resources, and the only one that is both malleable and language-agnostic. This makes it unique among the systems we reviewed, as it allows to create projections for very diverse use cases.

## 5 CASE STUDIES

Since LORGNETTE is a framework for creating special kinds of user interfaces, we follow Olsen's recommendations for evaluating such systems [54] and demonstrate how LORGNETTE fulfils Resnick et al.'s *low threshold, high ceiling, wide walls* goals [67]. To this end, we describe how we used LORGNETTE to implement projections in five different use cases. We present the situations and the projections, explain how we implemented them, and highlight key differences with what could have been done with other systems. All the projections are shown in Figure 2. We refer to them using numbers in yellow discs.

### 5.1 Manipulating colours

Colour pickers are one of the few projections that are found in several established programming systems. As an example, as of March 2023, colour pickers are available in professional software such as JetBrains' IntelliJ IDEA for Java and Microsoft's Visual Studio Code for CSS. Yet, even such a simple projection cannot be adapted to

[16] https://www.tablesgenerator.com/
[17] https://webcode.tools/generators/css

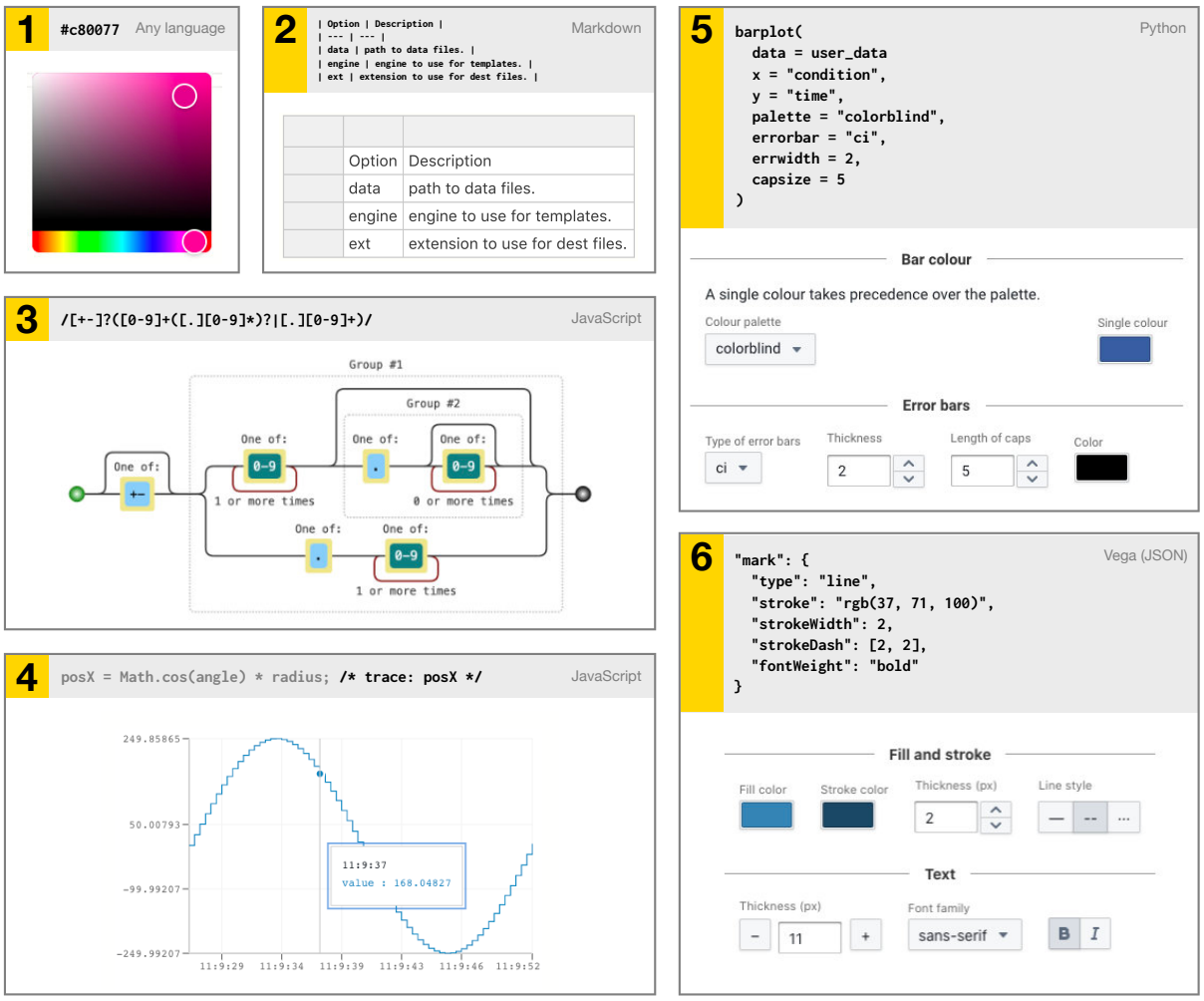



**Figure 2: The six projections we implemented to demonstrate LORGNETTE's capabilities along with the code fragments they represent. 1 A colour picker for hexadecimal colour codes. 2 An interactive Markdown table. 3 A railway diagram for regular expressions. 4 A 2D plot of values taken by the posX variable at runtime. 5 A form to configure Seaborn's barplot function. 6 A form to configure the style of Vega marks.**

work in other situations than the one they were hardcoded for: users have no way to make them work with languages and colour formats they were not designed for, even though their syntax may be very similar.

We used LORGNETTE to create a projection that lets users view and manipulate colours encoded as hexadecimal colour codes such as `#c80077` (1 in Figure 2). It makes it easy to manipulate any such sequence of text as a colour, no matter whether it is a primitive value, a string, a comment, etc. Since it only relies on text, it is automatically available in every language, including languages that LORGNETTE cannot parse.

To create this projection with LORGNETTE, we used a regular expression pattern template. This template allows to create slots for each named capture group of the regular expression. We created three slots, one for each successive pair of symbols following the initial # sign, that are each evaluated as numbers written in base 16. Finally, we added transformer functions that (1) wrap the RGB values into an object at the end of the forward mapping, to create a model with a `color` field, which is what the colour picker user interface expects; and (2) unwrap the modified `color` object at the beginning of the backward mapping, which is what the template expects.

```
const hexadecimalEvaluator = new NumericEvaluator({
    isIntegerValue: true,
    integerBase: 16
});

const template = new RegexTemplate(
    "#(?<r>[a-fA-F0-9]{2})(?<g>[a-fA-F0-9]{2})(?<b>[a-fA-F0-9]{2})",
    {
        "r": hexadecimalEvaluator,
        "g": hexadecimalEvaluator,
        "b": hexadecimalEvaluator
    },
```

```
    {
        transformTemplateModel: model => {
            return { color: model };
        },
        transformUserInterfaceModel: model => {
            return { ...model.color };
        }
    }
);

registerProjection({
    name: "Hexadecimal colour picker",
    ...template.resourcesAndMappings,
    userInterface: "color-picker",
    renderer: "side"
});
```

The projection could be adapted to support 3- and 8-digits hexadecimal colours, as well as other common notations such as `rgb(200, 0, 119)` and `hsl(324, 1.0, 0.39)`. This would only require using templates with slightly different regular expressions and adding conversions between colour spaces in the transformer functions when needed. The only other system to support this type of language-agnostic textual patterns is Codelets, but since it only works with predefined code snippets, it could not recognise hexadecimal codes that are already written, e.g., in existing code bases or in code pasted from the internet.

## 5.2 Authoring tables

Tables written in languages such as HTML, Markdown and LaTeX are notoriously tedious to manipulate as text. As an example, in all of these languages, cells are grouped by row, which means that inserting, reordering or deleting a column requires as many edits as the number of rows. In reaction to the need for a more interactive interface to create and modify such tables, authors have reported using spreadsheets to organise their data beforehand [30] and online code generators to synthesise code in the appropriate language. However, these strategies also increase the time needed to author these tables and the workload induced by the repetitive context switches between multiple programs every time the table must be modified.

We used Lorgnette to create a projection for Markdown tables (2 in Figure 2). It mimics the type of feature offered by the aforementioned code generators: cells can be edited by double clicking on them, and rows and columns can be inserted and deleted using a contextual menu and moved by dragging their headers. To create the projection, we used a syntactic pattern to target nodes representing tables. We then iterate over the row and cell nodes to collect the content of every cell into a two dimensional array to create the appropriate model in the forward mapping, and do the exact opposite to replace the table's content with the new model in the backward mapping.

Projections for manipulating tables have been shown in mage and Livelits, but they were designed to interact with tables which only exist at runtime, such as dataframes created by reading a CSV file, whereas authoring tables written in document description languages usually requires to modify tables written in the code. Software such as Adobe Dreamweaver also let users interact with HTML tables in a WYSIWYG fashion, but it is specialised for a single language and requires to interact with the formatted output instead of letting users focus on the content and the structure of the table. In contrast, Lorgnette allowed us to easily create a tool to write and edit tables written in Markdown, akin to *i*-LaTeX's projection for LaTeX tables [30]. Since tables usually have a clear hierarchy in syntax trees, creating similar projections for other document description languages would likely be as straightforward as it was for Markdown.

## 5.3 Writing regular expressions

Regular expressions are a powerful mechanisms for specifying textual patterns, but they must often be written in an embedded domain-specific language, e.g., as a literal value or as a string, making them hard to use. Various techniques have been proposed to help users write regular expressions, ranging from synthesising them from positive and negative examples [84] to explaining their meaning using augmented code editors [8]. Yet, these techniques often remain research prototypes or only exist as independent web services such as Regexr[18] and Regulex[19] which, just like code generators for tables, force programmers to switch to a different program every time they want to author or analyse a regular expression.

We used Lorgnette to create projections that recognise regular expressions in the code and add a button next to them to visualise them as railway diagrams (3 in Figure 2). We created two projections for JavaScript: one for regular expression literals, e.g., `/ab*/g`, and one for the arguments of the regular expression constructor, e.g., `new RegExp("ab*", "g")`. We used two syntactic patterns to target the two contexts these regular expressions can appear in and isolated the expressions by computing the appropriate substrings in the forward mappings.

```
const regexLiteralPattern = new SyntacticPattern(node =>
    node.type === "RegularExpressionLiteral"
);

const regexConstructorPattern = new SyntacticPattern(node =>
    node.type === "NewExpression" &&
    node.childNodes[1].text === "RegExp"
);
```

In each projection, the user interface displays the railway diagram created by Regulex. It works by loading the Regulex website in an `<iframe>` element that loads a dynamically constructed URL containing the regular expressions to visualise. Since Regulex does not allow to modify the regular expression by interacting with the diagram, our projection does not either. Nonetheless, we believe this example shows how Lorgnette allows to embed online tools in a code editor in a contextually relevant fashion, therefore reducing the number of context switches, even if their source code is not publicly available. In addition, since most languages use a similar regular expression dialect, the projection could be reused to visualise (most) regular expressions written in other programming languages simply by changing the pattern to search for in the code.

## 5.4 Tracing variables at runtime

Inspecting the current value of a variable is a staple of debugging. Displaying the values of the variables in the current namespace is a

[18] https://regexr.com/
[19] https://jex.im/regulex

standard feature in many debuggers, and systems such as Light Table[20] and Google Chrome's debugger can even display the value of expressions next to where they appear in the code. In addition, even when debuggers are available, many programmers also rely on print statements that they manually add to their code. Yet, none of these techniques allows to visualise how the value of a variable evolves over time or to compare it to previous values. At best, programmers are left with a sequence of raw values printed in the console that they must carefully analyse.

Using LORGNETTE, we developed a projection that serves as a debugging tool that plots the values taken by a variable using a 2D plot user interface (4 in Figure 2). The projection is designed for JavaScript. It works with comments that start with `trace:` followed by the name of the variable whose value must be plotted. They are targeted using a regular expression pattern. Using the fragment, a runtime query is then constructed by extracting the variable name to evaluate from the comment's body. Finally, the forward mapping maps the runtime response (which are automatically accumulated by the projection) to a model formed by a list of objects containing the values of the variable and the time at which it was modified.

```
const pattern = new RegexPattern(
    "/\\*\\s*trace\\s*:\\s*\\w+\\s*\\*/"
);

const runtimeRequests = ({ fragment }) => {
    const text = fragment.text;
    const identifier = text
      .slice(text.indexOf(":") + 1, text.length - 2)
      .trim();

    return [new RuntimeRequest(fragment, identifier)];
});

const forwardMapping = ({ runtimeResponses }) => {
    return {
      values: runtimeResponses.map(response => {
        return {
          value: Number(response.content),
          timestamp: response.receptionTime
        };
      })
    };
});
```

We tested this projection by connecting the VSC code editor augmented with LORGNETTE to a Chromium instance that was running a script that made an element move in a circular pattern. We used the projection to trace the values of the variables representing the X and Y positions of the element after each update. Figure 2 shows the result of plotting its X positions. We are not aware of any such projection for imperative programs in the systems we reviewed, but we were inspired by the visual probes of Poker [16], which can dynamically visualise the evolution of values in the graph of a reactive program.

## 5.5 Configuring lists of properties

It is quite common to specify a number of properties by hand when programming. This typically happens when configuring the graphical properties of a drawing primitive or a plotting function; when editing a configuration file or the front-matter of a document; as well as when configuring advanced tools such as Pandoc and FFmpeg, as seen in the results of the design workshop (R5, D10, D4). Since there is no systematic way to guess what properties can be configured and which values can each of these properties take, it usually requires to switch back and forth between the code and the appropriate documentation. This may be a daunting task when there are several dozens of optional properties, as illustrated by, e.g., Matplotlib's `plot` function in Python.[21]

Forms are convenient tools for configuring properties without knowing what exactly is feasible in advance. We therefore used LORGNETTE to create two custom projections with a form user interface: one for the `barplot` function of the Seaborn library[22] in Python, and one for the style properties of Vega marks [70] in JSON (5 and 6 in Figure 2). For each projection, we used a template with one slot per property of interest. For the user interfaces, we took advantage of the fact that LORGNETTE's forms can be entirely customised by describing their content using JSX elements. Form elements can be automatically bound to a property of the model. Forms can include other kinds of elements too, such as text and images, let the user customise the style of the form elements, etc.

```
const template = new PythonFunctionCallArgumentsTemplate(
    "barplot",
    {
      "palette": new StringEvaluator(),
      "color": new ColorEvaluator(),
      "errorbar": new StringEvaluator(),
      "errcolor": new ColorEvaluator(),
      "errwidth": new NumericEvaluator(),
      "capsize": new NumericEvaluator()
    }
);

const formContent = <>
    <Section title="Bar colour">
      <p>A single colour takes precedence over the palette.</p>
      <Select
        formEntryKey="palette"
        label="Colour palette"
        items={["deep", "muted", "pastel", "dark", "colorblind"]}
      />
      <ButtonColorPicker
        formEntryKey="color"
        label="Unique colour"
        defaultValue={Color.fromHexString("#3C5CA0")}
      />
    </Section>
    <Section title="Error bars">
      <Select
        formEntryKey="errorbar"
        label="Type of error bars"
        items={["None", "ci", "pi", "se", "sd"]}
      />
      <NumberInput
        formEntryKey="errwidth"
        label="Thickness"
      />
      <NumberInput
        formEntryKey="capsize"
        label="Length of caps"
      />
      <ButtonColorPicker
        formEntryKey="errcolor"
        label="Color"
      />
    </Section>
</>;
```

[20] http://lighttable.com/

[21] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

[22] https://seaborn.pydata.org

```
registerProjection({
    name: "Barplot configurator",
    requirements: { language: ["python"] },
    ...template.resourcesAndMappings,
    userInterface: {
      type: "form",
      options: { content: formContent }
    },
    renderer: "side"
});
```

These forms support a form of exploratory programming, as they allow to try and compare different configuration options (such as different colours and thicknesses) until the output of the code is satisfying. By combining them with a continuous evaluation of the code, they can be used to create a live programming environment with projections offering interfaces similar to those of style inspectors in WYSIWYG interfaces. Projections that display customisable forms were previously demonstrated in Codelets [58], but the form elements could only be bound to regions of predefined textual patterns. They also appear in specialised systems, such as Ivy's custom templating languages for Vega [50], but they cannot be customised nor adapted to other languages.

## 6 LIMITATIONS AND FUTURE WORK

We demonstrated that LORGNETTE is versatile enough to create projections that help working with code in various languages and contexts. In the rest of this section we present the technical limitations of LORGNETTE that we have identified as well as directions for future work.

While we believe that most of the 62 projections presented in Table 1 could be readily recreated using LORGNETTE, we identified a few technical limitations in the current prototype. One such limitation is that LORGNETTE can only search for fragments in one file at a time. This prevents LORGNETTE from supporting composite code fragments formed by multiple code fragments split across several files, which are required to modify a CSS file by interacting with a table described in a HTML file (R9) or create and grade code assignments where questions and answers are located in separate files (D14, D16, R2, R3). Another limitation is that LORGNETTE is more tailored for projections of restricted pieces of code and local runtime data rather than entire programs and global runtime data, such as displaying the stack and the heap [33] or the call graph [45] of a program. We also did not address the challenge of creating a renderer that embeds projections in other documents as *i*-LaTeX does with PDFs generated from the code [30], therefore preventing the creation of projections for manipulating images directly in the output (D13). This leaves room for future opportunities to link code with locations in other documents, possibly inspired by existing solutions such as SyncTeX [46] and Source Maps.[23]

In addition to technical limitations, our work leaves a number of empirical questions open for future work. While our application of LORGNETTE to five cases shows that it is *possible* to use it to create different sorts of projections, it does not evaluate *how hard* it is. We posit that the main challenge for creating projections is to write the mappings. While it can be short and straightforward in some cases, e.g., when converting a colour to a different colour space, it can also be long and complex, e.g., when statically analysing the lifetime of variables in Rust [1] or the dependencies between iterations in parallelised loops in C [85]. Several techniques may be used to help write such mappings. Besides manually writing the functions' code, users may rely on templates to create projections in a declarative fashion, as in Vega-Lite [69] and Varv [11], or use code synthesis tools such as GitHub Copilot[24] to assist them in the task. Future work may explore which strategies work best by studying how users adapt existing projections, e.g., to work with a different syntax or language, and create new projections from scratch.

As the use of artificial intelligence techniques by programmers is on the rise, as shown by ChatGPT[25] being used to write, explain, debug and optimise code [60], so is the importance of making sure programmers remain able to understand and verify the code generated by such tools. Future work may study how projections could help programmers who use code synthesis tools detect faulty code or fine-tune values interactively with the help of appropriate user interfaces. Furthermore, we also left the collaborative aspect of writing code out of the discussion. This leaves room for studying how programmers would like to use and share projections in collaborative workspaces such as Codestrates [68], as well as other collaboration-specific issues that appear in such contexts.

## 7 CONCLUSION

We presented LORGNETTE, a new framework for creating code editing environments in which users can use and create projections to view and manipulate fragments of code. It is the first framework to equally support textual, syntactic and runtime resources and one of the few that works with multiple computer languages. We motivated its purpose by analysing 62 projections sourced from the literature and a design workshop, and we demonstrated its capabilities by implementing projections for five different situations. We also identified several limitations of our approach and directions for future work.

Each paradigm for authoring projections has advantages and weaknesses, including LORGNETTE. Yet, allowing end-users to create and appropriate projections is essential to spread the usage of semantic interaction. The success of open-source software and of the internet owes in great part to the freedom they offer users. The variety of computer languages also demonstrates the need for diversity in programming tools. We believe this also applies to projections and that making them malleable and multilingual is critical to their success.

## ACKNOWLEDGMENTS

We thank Alexandre Battut and Wendy Mackay for their help on organising the design workshop and the participants for their time. This work was partially supported by European Research Council (ERC) grant n°695464 ONE: Unified Principles of Interaction.

[23] https://developer.chrome.com/blog/sourcemaps/

[24] https://github.com/features/copilot

[25] https://chat.openai.com/

# REFERENCES


[1] Marcelo Almeida, Grant Cole, Ke Du, Gongming Luo, Shulin Pan, Yu Pan, Kai Qiu, Vishnu Reddy, Haochen Zhang, Yingying Zhu, and Cyrus Omar. 2022. RustViz: Interactively Visualizing Ownership and Borrowing. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC53370.2022.9833121

[2] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. In *Proceedings of the ACM on Programming Languages*, Vol. 4. 1–28. https://doi.org/10.1145/3428290

[3] Ian Arawjo. 2020. To Write Code: The Cultural Fabrication of Programming Notation and Practice. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. ACM, 1–15. https://doi.org/10.1145/3313831.3376731

[4] Ian Arawjo, Anthony DeArmas, Michael Roberts, Shrutarshi Basu, and Tapan Parikh. 2022. Notational Programming for Notebook Environments: A Case Study with Quantum Circuits. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22)*. ACM, 1–20. https://doi.org/10.1145/3526113.3545619

[5] Dimitar Asenov. 2017. *Envision: Reinventing the Integrated Development Environment.* Ph. D. Dissertation. ETH Zurich. https://doi.org/10.3929/ETHZ-A-010863881

[6] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 752–763. https://doi.org/10.1145/3180155.3180156

[7] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (2017), 72–80. https://doi.org/10.1145/3015455

[8] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. 2014. RegViz: Visual Debugging of Regular Expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 504–507. https://doi.org/10.1145/2591062.2591111

[9] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. 2013. *Deep into Pharo.* Square Bracket Associates. 420 pages.

[10] Alan F. Blackwell. 2018. A Craft Practice of Programming Language Research. In *Proceedings of PPIG 2018 — The 29th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2018)*. 1–9.

[11] Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandsted Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. In *CHI Conference on Human Factors in Computing Systems (CHI '22)*. ACM, 1–20. https://doi.org/10.1145/3491102.3502064

[12] Frederick Brooks. 1987. No Silver Bullet – Essence and Accident in Software Engineering. *Computer* (1987), 16.

[13] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474. https://doi.org/10.1561/1900000006

[14] Matthew Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the 31st Symposium on User Interface Software and Technology - UIST '18*. ACM, 977–989. https://doi.org/10.1145/3242587.3242600

[15] J. H. Connolly and D. J. Cooke. 2004. The Pragmatics of Programming Languages. *Semiotica* 2004, 151 (2004), 149–161. https://doi.org/10.1515/semi.2004.065

[16] Cloé Descheemaeker, Sam Van den Vonder, Thierry Renaux, and Wolfgang De Meuter. 2021. Poker: Visual Instrumentation of Reactive Programs with Programmable Probes. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. ACM, 14–26. https://doi.org/10.1145/3486605.3486785

[17] E.W. Dijkstra. 1977. Programming : From Craft to Scientific Discipline. In *Proceedings 5th International Computing Symposium (Liège, Belgium, April 4-7, 1977)*, E. Morlet and D. Ribbens (Eds.). North-Holland Publishing Company, 23–30.

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

[19] Alan Dix. 2007. Designing for Appropriation. In *Proceedings of HCI 2007 The 21st British HCI Group Annual Conference University of Lancaster (21)*. 1–4.

[20] Jonathan Edwards. 2005. Subtext: Uncovering the Simplicity of Programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, 505–518. https://doi.org/10.1145/1094811.1094851

[21] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. *arXiv:1707.09627 [cs]* (2018). arXiv:1707.09627 [cs]

[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, 391–406. https://doi.org/10.1145/2048066.2048099

[23] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Vol. 8225. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[24] M. Erwig and B. Meyer. 1995. Heterogeneous Visual Languages-Integrating Visual and Textual Programming. In *1995 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Comput. Soc. Press, 318–325. https://doi.org/10.1109/VL.1995.520825

[25] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 153:1–153:29. https://doi.org/10.1145/3485530

[26] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems* 29, 3 (2007), 17–es. https://doi.org/10.1145/1232420.1232424

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[28] Joseph (Yossi) Gil and Itay Maman. 2005. Micro Patterns in Java Code. In *Proceedings of the 20th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA '05*. ACM, 97. https://doi.org/10.1145/1094811.1094819

[29] Yossi Gil, Ori Marcovitch, and Matteo Orrú. 2019. A Nano-Pattern Language for Java. *Journal of Computer Languages* 54 (2019), 100905. https://doi.org/10.1016/j.cola.2019.100905

[30] Camille Gobert and Michel Beaudouin-Lafon. 2022. *i-*LaTeX: Manipulating Transitional Representations between LaTeX Code and Generated Documents. In *CHI Conference on Human Factors in Computing Systems (CHI '22)*. ACM, 1–16. https://doi.org/10.1145/3491102.3517494

[31] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley.

[32] T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174. https://doi.org/10.1006/jvlc.1996.0009

[33] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th Technical Symposium on Computer Science Education - SIGCSE '13*. ACM, 579–584. https://doi.org/10.1145/2445196.2445368

[34] Han L. Han, Miguel A. Renom, Wendy E. Mackay, and Michel Beaudouin-Lafon. 2020. Textlets: Supporting Constraints and Consistency in Text Documents. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems - CHI'20*. ACM, 1–13. https://doi.org/10.1145/3313831.3376804

[35] Keith Hanna. 2002. Interactive Visual Functional Programming. In *Proceedings of the 7th International Conference on Functional Programming - ICFP '02*. ACM, 145–156. https://doi.org/10.1145/581478.581493

[36] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Symposium on User Interface Software and Technology - UIST'19*. ACM, 281–292. https://doi.org/10.1145/3332165.3347925

[37] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. ACM, 654–664. https://doi.org/10.1145/3180155.3180165

[38] Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. 2021. Natural Language-Guided Programming. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2021)*. ACM, 39–55. https://doi.org/10.1145/3486607.3486749

[39] Joshua Horowitz and Jeffrey Heer. 2023. Live, Rich, and Composable: Qualities for Programming Beyond Static Text. https://doi.org/10.48550/arXiv.2303.06777 arXiv:2303.06777 [cs]

[40] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (2023), 1–59. https://doi.org/10.22152/programming-journal.org/2023/7/13

[41] Stephen Kell. 2017. Some Were Meant for C: The Endurance of an Unmanageable Language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, 229–245. https://doi.org/10.1145/3133850.3133867

[42] 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 Symposium on User Interface Software and Technology - UIST'20*. ACM, 140–151. https://doi.org/10.1145/3379337.3415842

[43] 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 2006 CHI Conference on Human Factors in Computing Systems - CHI '06*. ACM, 387–396. https://doi.org/10.1145/1124772.1124831

[44] Michael Kölling, Neil Brown, and Amjad Altadmri. 2017. Frame-Based Editing. *Journal of Visual Languages and Sentient Systems* 3, 1 (2017), 40–67. https://doi.org/10.18293/VLSS2017-009

[45] Thomas D. LaToza and Brad A. Myers. 2011. Visualizing Call Graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 117–124. https://doi.org/10.1109/VLHCC.2011.6070388

[46] Jérôme Laurens. 2008. Direct and Reverse Synchronization with SyncTeX. *TUGBoat* 29, 3 (2008), 365–371.

[47] Richard Lin, Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Bjoern Hartmann. 2021. Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. ACM, 1039–1049. https://doi.org/10.1145/3472749.3474804

[48] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28. https://doi.org/10.1145/3276497

[49] Michael J. McGuffin and Christopher P. Fuhrman. 2020. Categories and Completeness of Visual Programming and Direct Manipulation. In *Proceedings of the ACM International Conference on Advanced Visual Interfaces - AVI'20*. ACM, 1–8. https://doi.org/10.1145/3399715.3399821

[50] Andrew M McNutt and Ravi Chugh. 2021. Integrated Visualization Editing via Parameterized Declarative Templates. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. ACM, 1–14. https://doi.org/10.1145/3411764.3445356

[51] Gina Neff and David C. Stark. 2002. Permanently Beta: Responsive Organization in the Internet Era. *Institute for Social and Economic Research and Policy Working Papers* (2002). https://doi.org/10.7916/D8G44X47

[52] James Noble and Robert Biddle. 2002. Notes on Postmodern Programming. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*.

[53] Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books.

[54] Dan R. Olsen. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, 251–258. https://doi.org/10.1145/1294211.1294256

[55] Cyrus Omar and Jonathan Aldrich. 2018. Reasonably Programmable Literal Notation. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 106:1–106:32. https://doi.org/10.1145/3236801

[56] 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*. ACM, 511–525. https://doi.org/10.1145/3453483.3454059

[57] Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *Proceedings of the 34th International Conference on Software Engineering - ICSE'12*. IEEE, 859–869. https://doi.org/10.1109/ICSE.2012.6227133

[58] Stephen Oney and Joel Brandt. 2012. Codelets: Linking Interactive Documentation and Example Code in the Editor. In *Proceedings of the 2012 CHI Conference on Human Factors in Computing Systems - CHI '12*. ACM, 2697. https://doi.org/10.1145/2207676.2208664

[59] Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked Visualisations via Galois Dependencies. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498668

[60] Jeffrey M. Perkel. 2023. Six Tips for Better Coding with ChatGPT. *Nature* 618, 7964 (2023), 422–423. https://doi.org/10.1038/d41586-023-01833-0

[61] Tomas Petricek. 2020. Foundations of a Live Data Exploration Environment. *The Art, Science, and Engineering of Programming* 4, 3 (2020), 8. https://doi.org/10.22152/programming-journal.org/2020/4/8

[62] Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 155–174. https://doi.org/10/ghs5sn

[63] 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 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[64] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2018), 1. https://doi.org/10.22152/programming-journal.org/2019/3/1

[65] Miguel A. Renom, Baptiste Caramiaux, and Michel Beaudouin-Lafon. 2022. Exploring Technical Reasoning in Digital Tool Use. In *CHI Conference on Human Factors in Computing Systems (CHI '22)*. ACM. https://doi.org/10.1145/3491102.3501877

[66] 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 (2009), 60–67. https://doi.org/10.1145/1592761.1592779

[67] Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, and Mike Eisenberg. 2005. Design Principles for Tools to Support Creative Thinking. *Report of Workshop on Creativity Support Tools* 20 (2005).

[68] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Symposium on User Interface Software and Technology - UIST '17*. ACM, 715–725. https://doi.org/10.1145/3126594.3126642

[69] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[70] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 659–668. https://doi.org/10.1109/TVCG.2015.2467091

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

[72] Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. 2017. *Transmorphic: Mapping direct manipulation to source code transformations*. Number 110 in Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Universitätsverlag.

[73] Ben Shneiderman and Richard Mayer. 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Sciences* 8, 3 (1979), 219–238. https://doi.org/10.1007/BF00977789

[74] Charles Simonyi. 1995. *The Death Of Computer Languages, The Birth of Intentional Programming*. Technical Report MSR-TR-95-52. Microsoft Research.

[75] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, 451–464. https://doi.org/10.1145/1167473.1167511

[76] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual Augmentation of Source Code Editors: A Systematic Mapping Study. *Journal of Visual Languages & Computing* 49 (2018), 46–59. https://doi.org/10.1016/j.jvlc.2018.10.001

[77] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

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

[79] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. 2019. Lessons Learned from Developing Mbeddr: A Case Study in Language Engineering with MPS. *Software & Systems Modeling* 18, 1 (2019), 585–630. https://doi.org/10.1007/s10270-016-0575-4

[80] Markus Voelter and Sascha Lisson. 2014. Supporting Diverse Notations in MPS' Projectional Editor. In *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages*. 7–16.

[81] Susan Wiedenbeck. 1986. Beacons in Computer Program Comprehension. *International Journal of Man-Machine Studies* 25, 6 (1986), 697–709. https://doi.org/10.1016/S0020-7373(86)80083-9

[82] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Symposium on User Interface Software and Technology - UIST'20*. ACM, 152–165. https://doi.org/10.1145/3379337.3415851

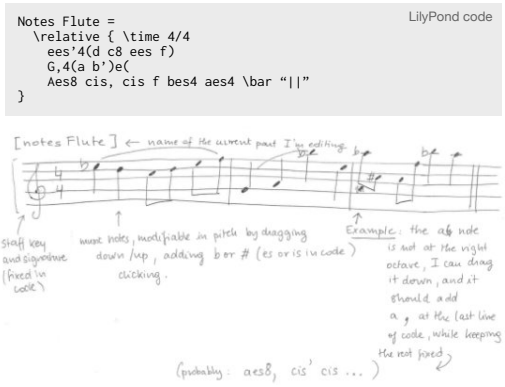
[83] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd Symposium on User Interface Software and Technology - UIST'20*. ACM, 126–139. https://doi.org/10.1145/3379337.3415890

[84] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Symposium on User Interface Software and Technology - UIST'20*. ACM, 627–648. https://doi.org/10.1145/3379337.3415900

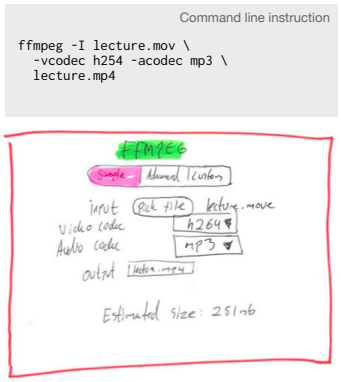
[85] Oleksandr Zinenko, Cédric Bastoul, and Stéphane Huot. 2015. Manipulating Visualization, Not Codes. In *International Workshop on Polyhedral Compilation Techniques (IMPACT)*. 1–8.

# A DESIGN WORKSHOP RESULTS

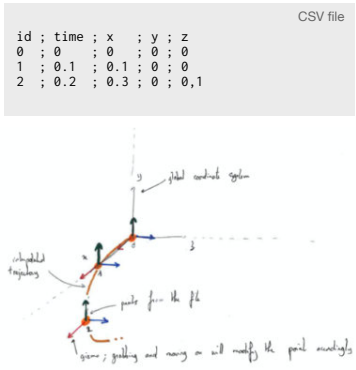
Table 3 (next page) lists the designs and redesigns that were created by the participants of the design workshop, while Figure 3 shows examples of code snippets and sketches that were created by workshop participants.




**(a) Design D1.**




**(b) Design D4.**




**(c) Design D8.**

**Figure 3: Examples of projections created by workshop participants that include the sample code (top) and the sketch (bottom).**

# B PAPERS WITH PROJECTIONS

Table 4 lists the 22 systems that implement projections that we collected from the literature We collected 39 projections from these systems. The table only contains 37 concepts, as two systems allow to project the same concept in two different ways: state machines can be projected either as a grid or as a graph in MPS [79], and dataframes can be projected either as a grid or as 2D plot in mage [42].

**Table 4: List of the systems containing projections that we collected from the literature.**

| System | Projected concepts |
|---|---|
| Livelits [56] | Colour<br>Dataframe<br>Image transformation |
| Palettes [57] | Colour<br>Regular expression |
| Codelets [58] | GUI component properties<br>Graphical properties<br>Animation properties |
| Racket's visual macros [2] | Form<br>Tree<br>Tsuro |
| Heterogeneous languages [24] | List<br>Tree<br>State machine |
| Vital [35] | List |
| Alectryon [62] | Tree<br>Coded formula<br>Conway's *Game of Life* |
| MPS [79] | State machine |
| RustViz [1] | Rust's ownership |
| Python Tutor [33] | Runtime stack and heap |
| Poker [16] | Reactive streams |
| Reacher [45] | Call graph |
| mage [42] | Dataframe<br>Image transformation |
| The Gamma [61] | Dataframe |
| *i*-LaTeX [30] | Document table<br>Grid layout<br>Coded formula<br>Image transformation |
| B2 [82] | Dataframe |
| Skyline [83] | Machine learning metrics |
| Clint [85] | Loop parallelisation |
| Barista [43] | Coded formula |
| Visual HDL blocks [47] | Hardware circuit |
| Notate [4] | Quantum circuit |
| RxFiddle [6] | Reactive stream |

**Table 3: Description of the designs (prefixed with D) and redesigns (prefixed with R) created by the participants of the design workshop. Rows in grey indicate the designs that we ignored in our analysis (see the text for justifications).**

| N° | Description of the design |
|---|---|
| D1 | An interactive music staff to edit LilyPond music scores. |
| D2 | A textual explanation of a binary optimisation program written in Python with Google's OR-tools. |
| D3 | A form to write and edit a configuration file for, e.g., connecting to a database or a SMTP server. |
| D4 | A form to configure the options of the `ffmpeg` command line utility, e.g. audio and video output parameters. |
| D5 | An interface to paste code that was previously copied and matches the expected type at the cursor position. |
| D6 | A text editor to modify a Javadoc comment in the code from within the generated documentation. |
| D7 | A graph whose nodes represent functions that can be edited as text and arcs represent calls between functions. |
| D8 | A 3D trajectory editor for a CSV file whose rows contain timestamps and 3D coordinates. |
| D9 | Previews of files (from their paths) and patterns that will be expanded when typing a command in a terminal. |
| D10 | A form to configure a list of properties passed to a function to use the Pandoc conversion tool in Swift. |
| D11 | A grid that represents a webpage and allows to create HTML elements spanning over the selected cells. |
| D12 | A map to draw trajectories and interact with a distribution defined along them to generate data in a table. |
| D13 | An interactive image that can be moved and resized in a PDF generated from LaTeX code. |
| D14 | A list of regions in the code that can be hidden/shown to generate Python code to be completed by students. |
| D15 | An interface to syntactically highlight SQL requests written as strings in Java. |
| D16 | An interface to replace code regions by "TODO" comments to create code assignments for students. |
| D17 | A sequence diagram that shows property accesses (horizontal arrows) between objects (vertical lines). |
| R1 | A 3D space showing the evolution of the reference point and how it affects transformations in Processing. |
| R2 | An interface to show the expected solution of a coding assignment next to the student's code. |
| R3 | An interface to replace code regions in different ways to create code assignments with different levels of difficulty. |
| R4 | A typeset mathematical formula that can be edited to modify mathematics written in LaTeX. |
| R5 | A form to configure the fields of a front matter of a Markdown document written in YAML. |
| R6 | A text input to globally rename the "id" property of an element in HTML that is also used in CSS rules. |
| R7 | A preview of the triangles/rectangles formed by successive 3D coordinates in an OBJ file. |
| R8 | Previews of the definition of certain expressions at the location where they are used in the code. |
| R9 | A grid that allows to configure a grid layout to be applied to the children of an HTML element. |