No Grammar to Rule Them All:
A Survey of JSON-style DSLs for Visualization

Andrew M. McNutt

Abstract— There has been substantial growth in the use of JSON-based grammars, as well as other standard data serialization languages, to create visualizations. Each of these grammars serves a purpose: some focus on particular computational tasks (such as animation), some are concerned with certain chart types (such as maps), and some target specific data domains (such as ML). Despite the prominence of this interface form, there has been little detailed analysis of the characteristics of these languages. In this study, we survey and analyze the design and implementation of 57 JSON-style DSLs for visualization. We analyze these languages supported by a collected corpus of examples for each DSL (consisting of 4395 instances) across a variety of axes organized into concerns related to domain, conceptual model, language relationships, affordances, and general practicalities. We identify tensions throughout these areas, such as between formal and colloquial specifications, among types of users, and within the composition of languages. Through this work, we seek to support language implementers by elucidating the choices, opportunities, and tradeoffs in visualization DSL design.

Index Terms—Visualization grammar, Survey, Declarative specification, Domain-Specific Languages

1 Introduction

Domain-specific languages (DSLs) represented in standard data serialization formats, such as JSON or YAML, are an increasingly common [63] interface for the specification of visualizations across an array of contexts and tasks. These restricted textual languages allow for the declarative specification of both static and interactive graphics in a systematic manner that can be manipulated both by humans, making them attractive for end-user programming, and computational agents, making them appealing for artificial intelligence applications [89]. This language style appears in a surprisingly large variety of tools and systems but is well exemplified by Vega [69] and Vega-Lite [67].

While it is sometimes derided for usability issues [17, 36, 44], this language style has a variety of benefits. DSLs which employ it can be *expressive*, allowing for the concise manipulation of complex specifications with minimal textual modification [69]. Many of these languages enhance the *explorability* of a space of possible programs by simple and fluid movement between instances. Their limited scope enables some specifications to be used *portably*, such that charts created in one platform (such as a GUI like Voyager [86]) can be used in another environment (such as in the Python-based Altair [78]).

Despite the popularity (Fig. 2) of this approach, there has been little detailed analysis of the characteristics of these systems. Pu et al. [63] highlight the need for additional study of visualization grammars, while Wongsuphasawat [84] surveyed the more general space of JavaScript (JS) visualization libraries. Although they are insightful, these works leave critical questions about these DSLs open: *What problems do they seek to solve? Who are they designed to serve?* or more generally *What design and implementation patterns are used in JSON-style DSLs?*

In this paper we answer these questions by surveying visualization DSLs represented in standard data serialization languages covering academic, industrial, and open source language efforts, yielding 57 distinct languages (Fig. 3). We analyze each of these DSLs across a variety of dimensions including the motivations for their design, relationships with other languages, and the conceptual models which are utilized. We identify five sets of concerns (Fig. 1) which are critical to JSON-style DSL design and highlight a corresponding set of tensions to be navigated, such as the tension between formal and colloquial models or the effect on the DSL caused by the interplay of different intended users. In doing so we note opportunities, tradeoffs, and open challenges. To aid this analysis we collected examples of each DSL, yielding 4395 programs, available in our interactive supplement.

While JSON-style DSLs have been usefully employed in a variety of visual analytics systems [50, 66, 86, 93], we believe that a firmer grasp of the design space of this language form will help future languages better address the highlighted design questions. Moreover, a language's affordances, abstractions, and models guide the types of expression that are made using it [56, 80]. Thus, a stronger foundation may open the door to new forms of analysis and expression.

2 Related Work

We will now locate our study within prior work on DSLs in general (and review relevant terminology) and visualization DSLs specifically.

2.1 Domain Specific Languages

DSLs are a type of programming language designed to facilitate particular tasks within a chosen domain. While this term is variably defined, DSLs are usually (but not always [17]) defined as languages that are unable to execute general computations, in exchange for specific declarative notation related to a domain of interest—properties that differentiate them from General Purpose Languages (GPLs). The database query language SQL, the browser-styling language CSS, and the markup language LaTeX are all familiar examples of this design approach. Van Deursen et al. [77] argue that DSLs are useful because they allow domain experts to operate within the notation of a given domain, and assert that they are typically concise, reusable, and self-documenting. They also note that DSLs carry a host of disadvantages including maintenance costs, learnability issues, and the danger of *language cacophony* [17] resulting from a preponderance of languages.

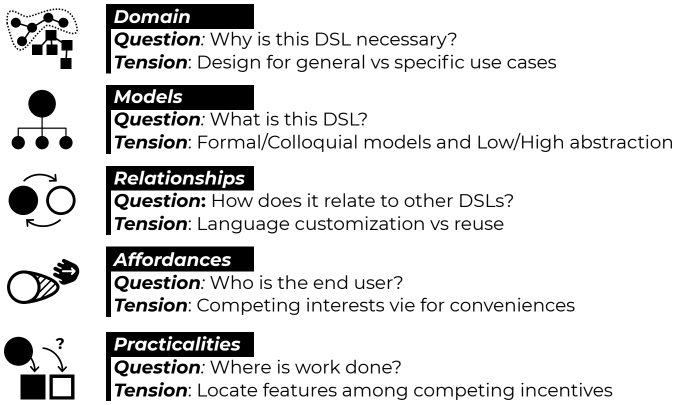
DSLs are often thought of as solely declarative, as the user specifies intent relative to the domain rather than through low-level details of how that action is executed. However, some DSLs use imperative syntax (e.g. shaders or Atlas [44]), but all DSLs in our study are declarative and the closest exceptions are pipeline models.

A common decomposition of DSLs [17, 53] describes them as *external* or *internal*. *External* languages define their syntax outside of their host language, such that they typically require separate parsing to execute. Some utilize custom syntax (e.g. SQL or CSS), while others elect to use standard data serialization grammars (e.g. XML or JSON).

This raises questions about what qualifies as a language as opposed to an API. Highlighting this ambiguity, Fowler [17] argues for a heuristic related to a fluent, composable, or language-like nature. This refers to the concept that "expressiveness comes not just from individual

Andrew M. McNutt is with University of Chicago. E-mail: mcnutt@uchicago.edu.

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx



Domain
Question: Why is this DSL necessary?
Tension: Design for general vs specific use cases

Models
Question: What is this DSL?
Tension: Formal/Colloquial models and Low/High abstraction

Relationships
Question: How does it relate to other DSLs?
Tension: Language customization vs reuse

Affordances
Question: Who is the end user?
Tension: Competing interests vie for conveniences

Practicalities
Question: Where is work done?
Tension: Locate features among competing incentives

Fig. 1. An overview of the analysis of our survey.

expressions, but also from the way they can be composed together" [17]. That is, there is a systemic way that the language operates, without needing special cases for every expression form. We use this heuristic to identify languages in our survey.

The complement to *external* DSLs are *internal* DSLs, which are embedded (as a library or through syntax extensions) into a host language—e.g. dplyr, d3, or RSpec. These languages provide expressivity similar to external languages, but do so in a way that confers the benefits and limitations of their host. Tobin-Hochstadt et al. [74] highlight the permeable border between languages and libraries in Racket, where libraries are distributed as language extensions—an ambiguity that is increasingly relevant as more visualization libraries adopt language-style interfaces. This architectural choice allows for the straightforward creation of richly expressive languages that are easy to integrate into a host, but can force constraints and notation which are inappropriate to the DSL domain. In contrast, Diderot [7] explicitly resists embedding so as to maintain the domain specificity of its type system.

Prior studies have sought to understand and typify DSL usage [3, 17, 61, 77]. Mernik et al. [53] describe design patterns exhibited at each stage of the DSL design process. Van Deursen et al. [77] characterize 75 DSLs by purpose. The analysis of our survey draws on these works but is designed to complement these considerations made by others by focusing on a particular domain. Erdweg et al. [16] identify a set of language composition mechanisms, which guides our discussion of the topic (although ours is adapted to a less-general domain). Several studies consider DSLs in specific domains, such as declarative data analytics [47], configuration languages [20], and visual computing [71]. Our work is related to these but is centered on visualization.

The use of standard serialization languages as carrier languages for DSLs is not new. XML and other hierarchical serialization languages have long been used as a way to configure applications [17, 20] and even specify visualizations (as in VizML [83]). Similarly, visualization is not unique in its use of JSON-DSLs. Beyond familiar uses such as configuration or NoSQL languages (e.g. MongoDB), they are used in domains as varied as statistical analysis [30], web development [2], narrative [8] and game generation [13], chatbots [35], dance [58], and fabrication [75]. We are interested in building a better understanding of JSON-style DSLs precisely because they are so prevalent—although their sudden prominence (Fig. 2) may indicate that they are a fad.

Despite their popularity, JSON-based languages are no panacea. They are sometimes maligned for their lack of programming usability features [36], rendering them hard to learn, debug, and extend [44]. These criticisms can be extended through the Cognitive Dimensions of Notations (CDN) [1], which are a suite of lightweight heuristics (*highlighted* throughout) that characterize the usability of notational interfaces. For instance, *viscosity* refers to the effort required to alter a program to a desired state, while *diffuseness* describes how terse the language is. This style of evaluation is especially useful for programming languages as it provides an external reference from which to critically reflect and a common grammar for usability issues. Thus, we can add to the criticisms of JSON DSLs by noting that they are subject to errors related to *premature commitment* (choices that make movement between states difficult), *hard mental operations* (the work required outside of the coding environment), and *progressive evaluation* (how incomplete programs are examined). Some serialization formats (e.g. YAML or JSON5) or languages (e.g. dhall [12]) seek to address usability issues—such as *diffuse* syntax and lack of a *secondary notation*. Yet, a consensus replacement has not materialized. This may be due to JSON's ubiquity in modern systems, which impart rich error handling and parsing, as well as typings via projects like JSON Schema [60].

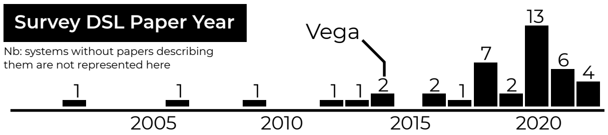


Fig. 2. Since **Vega**'s publication JSON-style DSLs have become popular.

2.2 Visualization DSLs

Visualization features a rich space in which a DSL can usefully abstract away unnecessary details in favor of domain-appropriate notation.

The most prominent visualization DSL is Wilkinson's Grammar of Graphics (GoG) [83], which describes the visualization process as a series of stages that results in a mapping of data attributes to visual-encoding channels (e.g. a penguin's flipper length mapped to spatial position). This approach allows the construction of myriad chart forms, in contrast to "chart templates" which map data attributes to aspects of a given chart type. GoG has influenced the development of many contemporary visualization language systems [4, 69, 82]. Friendly [18] reviews the model and its history.

Despite its prevalent use, the term "visualization grammar" is not well defined and is used in a variety of ways [63]. This term is sometimes used in the generative syntactic sense [57], referring to a system of rules that can be repeatedly applied to create particular shades of meaning. It may also be used to refer to composable systems of expression [42, 50], akin to Fowler's language definition. Further, it may specifically refer to variants of Wilkinson's [83] GoG. Still others use it to refer to any visualization system [88]. We do not strive to provide a conclusive definition of visualization grammars here, instead electing to use the slightly more general framing of DSLs—although we sometimes use the term to refer to a space of allowed syntax.

There are a wide variety of DSLs for visualization that fall outside of our language form of interest. The dot graph language and the mermaid diagramming language feature custom syntax for graph-based tasks. APT [46] is a DSL used to describe charts in a manner amenable to automated recommendations. Idyll [9] is a DSL for visualization-mediated explorable explanations. ViSlang [65] provides a system for making and coordinating small DSLs in SciVis, while Diderot [7] uses notation specifically aligned with the tensor-calculus operations which arise in that setting. Although the design patterns these DSLs manifest are valuable, they are beyond the scope of our study.

Several prior works study visualization languages. Wongsuphasawat [84, 85] sketched a taxonomy based on the level of abstraction covering graphics languages, low-level languages, grammars, high-level languages, and templating systems. Qin et al. [64] sketch a similar taxonomy based on an expressiveness-accessibility axis. We expand upon these studies through a more in-depth survey of narrower scope. Pu et al. [63] highlight the pressing need for more formal study of these entities. We seek to explore and address the questions they raise, as well as support future work by developing a richer understanding of the state of the art of this language form. Satyanarayan et al. [66] reflect on the design of visualization authoring systems, the results of which overlap with our study, although tuned to a slightly different domain.

3 Survey Methodology

We conducted a survey of visualization languages represented fully or partially in standard serialization languages (e.g. JSON, YAML, XML). This yielded 57 languages, which are displayed in Fig. 3.

We searched relevant academic search engines (Google Scholar, ACM Digital Library, IEEE Xplore) and code repositories (GitHub) for

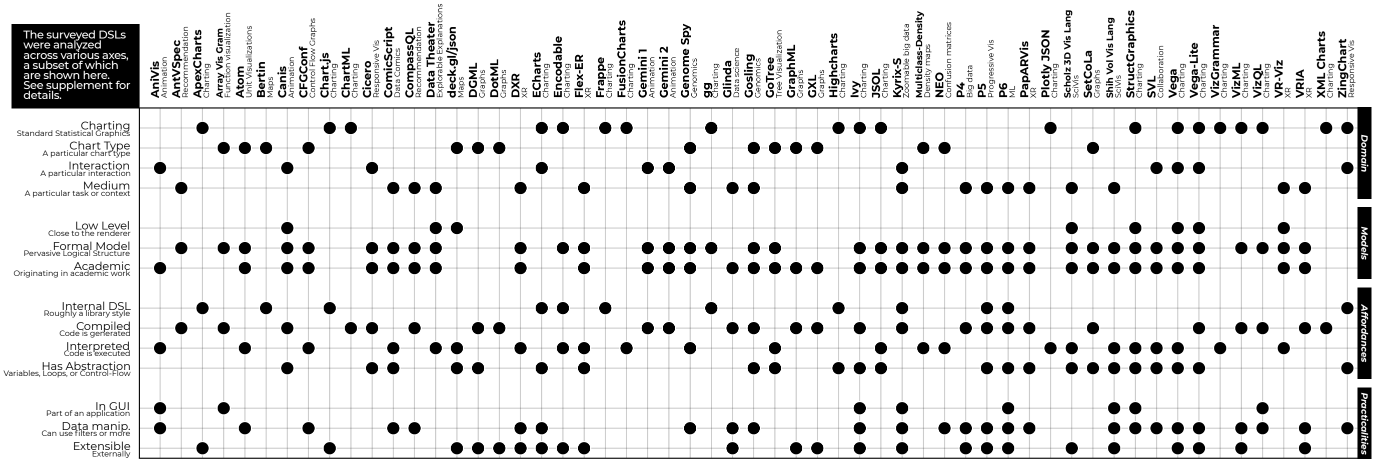


Fig. 3. DSLs select different feature combinations to achieve their goals. No one language, or feature combination, will suit all situations.

the following **keywords**: `DSL`, `domain-specific language`, `JSON`, `XML`, `YAML`, `visualization`, `map`, `grammar`, `language`, `chart`, and `graph`. Given the influence of the works on **Vega** and **Vega-Lite** on this type of DSL, we also reviewed all papers citing the papers documenting those systems [67–69]. We utilized snowball sampling whenever possible. We refer to systems in our survey like **Vega**, while we cite the works documenting them. See the appendix for a survey bibliography.

Our survey **criterion** included *any human-usable language that uses a standard serialization language to produce visualizations.* We follow Fowler's definition of a language [17] as being a system with a concept of composition or a sense of fluency. This *language nature* can manifest in a variety of ways, such as mark or series composition, as well as data or view algebras. We follow our prior definition [48] of a visualization as being a transformation of data meant to be interpreted by a human.

This criterion excludes some system types. SVG, HTML, and other high-level markup languages (as well as general-purpose JSON-based DSLs, such as Varv [2]) were excluded because they are capable of producing far more than visualizations. Also excluded were those merely subsetting another language—for instance, GraphScape [33] uses a non-interactive subset of **Vega-Lite** to explore sequence recommendation. These systems are excluded because they simply make use of a visualization DSL rather than constructing one. Systems that possess systematically described languages (e.g. Visception [37]) but either do not utilize a standard serialization language for its description or do not expose that language to the end-user were excluded. We focus on computer-based languages, which precludes natural language specifications such as in NL4DV [55]. While many libraries excluded by our criterion (e.g. ggplot) could be recast into JSON, we exclude them because we strive only to understand the patterns of those DSLs that have explicitly opted to use this representation—although the design of visualization APIs more generally is intriguing future work.

Examples. To facilitate comparison between the DSLs in our survey we collected representative samples of each language. For some DSLs this involved collecting every single example available (such as **Atom** and **Cicero**). For those with thriving communities (such as **Highcharts** and **Vega-Lite**), we only gathered examples from documentation or test repositories which provided sufficient examples for analysis. In some cases (e.g. **GoTree**) the only examples available were those found in the publications documenting those languages. While additional examples would always be useful, the samples collected were sufficiently representative to allow us to consider each of the axes of analysis. This yielded 4395 examples, although a small number of DSLs dominate this total. We present these materials as a DSL zoo [92] in our supplement at vis-json-dsls.netlify.app and for download at osf.io/e9v8y.

Analysis Process. We conducted an analysis seeking to answer: *What are the design and implementation patterns in visualization DSLs represented in standard serialization formats?* To do so, we analyzed each surveyed language across a set of topics. Our initial selection of topics was motivated by discussions of DSLs in general [17, 44, 53, 77], visualization DSLs [44, 84], as well as related work on visualization authoring systems [66]. We iteratively added and removed axes of analysis (analogous to codes) until a theoretical saturation was reached. Each axis was evaluated based on available documentation (such as a paper describing the DSL), the found examples for the language, or sometimes by reviewing the code itself. These results were grouped into categories (Fig. 1). See the supplement for details.

Our observations and analysis are descriptive, and not evaluative. Thus analyses such as locating DSLs within Satyanarayan et al.'s [66] expressiveness-learnability spectrum are beyond the scope of this work, as such comparisons would require experimental evaluation. We focus on patterns relevant to visualization DSLs and refer to exterior sources for those related to general DSL patterns [17, 31]. We forgo analysis of living sources (such as interviews) because these artifacts are sufficiently rich to conduct our analysis, although future work could be augmented by such explorations.

4 Analysis

We organize our discussion following our five concerns (Fig. 1) guided by the information-gathering interrogatives for each DSL.

4.1 Domain: Why is it necessary?

We begin by considering the aims of our DSLs in order to identify the problems they seek to solve and therein identify why they are necessary.

We found four purposes or domains for designing visualization DSLs: creating standard charts, creating a particular chart, enabling a specific interaction, and serving a certain task— as in Fig. 3. These various purposes highlight a critical tension: *why not just use something that already exists?* Indeed, many of the graphic types and domains can be addressed using **Vega** or GoG [18] with enough manipulation. This issue can be seen as a *Turing tar pit* [59] in which everything is possible, but nothing is easy. The value of using DSLs is exactly to avoid this pitfall: allowing some things to be easy by making some things (that are not relevant to the domain) impossible.

Standard charts. Most DSLs focused on standard charts, such as bar charts or scatter plots. Among these, the ostensible purpose varied, featuring different levels of abstraction, contexts, means of expression, or implementation affordances—each of which we discuss in subsequent sections. For instance, **Vega-Lite** enables standard charting tasks using a formal GoG-inspired approach, while **ECharts** employs a colloquial chart-type model that uses its close connection to the browser to provide responsive and progressive analytics features [40]. The *purpose* of such languages is then the additional affordances brought to the design.

Chart Forms. Some DSLs focus on enabling a particular chart form or genre. For instance, several languages support maps (such as **Bertin** and **deck.gl/json**), while others focus on graphs (**GraphML**, **GXL**, **CFGConf**, and **SetCoLa**). **NEO** enables confusion matrices. By focusing on a specific chart form a DSL can tailor its notation to the concerns of that graphic. To wit, graphs specify node position as relationships between entities and not in terms of spatial attributes. This allows languages like **GraphML** to focus on only high-level attributes—although adding notions of axial direction can be useful (which **SetCoLa** achieves by encoding directional and relative properties as constraints).

Some chart families—like Gantt charts, Sunbursts, HOP Plots, or Euler diagrams—can be constructed through visualization DSLs whose notations are not well aligned with those charts. For instance, while Gantt charts can be produced using general charting tools like **Vega-Lite** (Fig. 4), they do so in a fashion that is not well matched with the data. Gantt charts show the relationship between projects over time in a DAG structure, sometimes featuring graph-based computations such as *critical path* (maximum path length), neither of which are well supported in the tabular data model used in many DSLs. The result is that the user potentially needs to make tedious layout re-computations and round-trips to the data. While this data type and associated transformations could be embedded into **Vega-Lite**, doing so would add to the growing complexity of that language.

We suggest that the creation of task-specific DSLs (or what Guzdial and Shreiner [21] would call Teaspoon languages) that allow end-users to author charts in the vernacular of a particular chart or data form to be an important opportunity. **Encodable** addresses this problem through a **Vega-Lite**-inspired component abstraction over arbitrary chart forms—although this is done in such a way that binds the charts to the JS-implementation (precluding portability) and requires a tabular data structure (which can be at odds with the domain, as in Gantt charts).

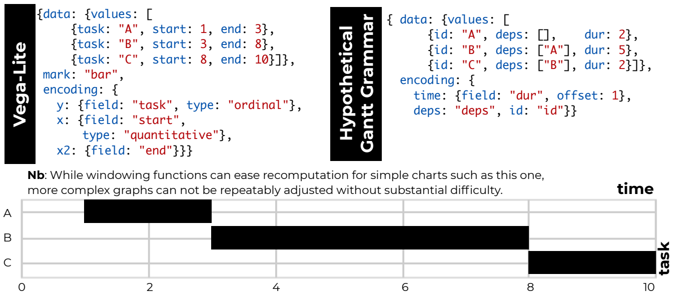
Interactions. Most languages in our survey produced interactive visualizations that supported at least some simple interactions, such as tooltips. However, some languages specifically focus on nuanced or uncommon interactions. Some DSLs center animation, such as **AniVis**, **Gemini 1/2**, and **Canis**. **Cicero**, **ECharts**, and **ZingChart** emphasize creating responsive visualizations. **SVL** supports some collaborative visual analytics interactions. Focusing on a particular interaction allows the language to surface attributes that are specific to that interaction, such as **Gemini 2**'s use of keyframes as a first-class element of the language.

Careful balancing of novel and expected features is essential, as having too many options can dilute the specificity of the DSL, however, missing anticipated features can ruin its domain utility. For instance, zooming is an interesting addition to abstract visualizations in **Gosling**, while in map-focused DSLs it is all but required.

A number of DSLs focus only on the interaction of concern and offload the remainder of the work to another DSL through compilation (Sec. 4.3). This style of interaction injection is a valuable component of some JSON-style DSLs. The restricted grammars allow simple language composition, allowing each DSL to do one thing well. Similar effects can be achieved with plugin architectures, however such methods are usually inaccessible to end-users.

Nearly all interactions in these DSLs are *transient*, with only one DSL (**Glinda**) supporting updates from their state into their specification. While some interfaces (e.g. B2 [91]) provide mechanisms to reify interactions into code, it is a missed opportunity that such *bidirectional interactions* are relegated to external tools given JSON-like DSLs computational malleability. For instance, tasks like annotation may be easier to complete using direct manipulation as it does not require repeated round trips to code [79]. We suggest that such tasks may be well-supported by DSLs that allow alteration of their specification from their output *in addition* to from the code, such that changes to the output are reflected in the input and vice-versa. Similar techniques have been employed to allow bidirectional updates to SVG drawings formed through functional programming languages [22, 24]. As JSON-style DSLs can be manipulated more easily than most GPLs, we suggest that future systems should explore this intriguing feature domain.

Mediums. The purpose of a number of systems is to enable use of a particular medium, form of data, or set of tasks. As noted in Fig. 3, these DSLs address a wide range of domains including SciVis, big data, and data science/ML. This broad purpose highlights that visualization tasks occur outside of the tidy small-data abstract-charting sandbox that visualization DSLs (and systems [66]) often target, and moreover, that conceptual adaptations can be usefully made to serve other use cases.



```
{data: {values: [
     {task: "A", start: 1, end: 3},
     {task: "B", start: 3, end: 8},
     {task: "C", start: 8, end: 10}]},
 mark: "bar",
 encoding: {
   y: {field: "task", type: "ordinal"},
   x: {field: "start",
       type: "quantitative"},
   x2: {field: "end"}}}
```

```
{ data: {values: [
      {id: "A", deps: [],    dur: 2},
      {id: "B", deps: ["A"], dur: 5},
      {id: "C", deps: ["B"], dur: 2}]},
  encoding: {
    time: {field: "dur", offset: 1},
    deps: "deps", id: "id"}}
```

Nb: While windowing functions can ease recomputation for simple charts such as this one, more complex graphs can not be repeatably adjusted without substantial difficulty.

Fig. 4. While these Gantt chart specifications are similar in length and complexity, the task-specific DSL does not require the user to manually update the positions after a data update.

Focusing on domain allows selection of appropriate notation. For instance, some domains use their own coordinate systems either by convention or necessity. **Genome Spy** and **Gosling** include an idea of genomic coordinates. **ComicScript** has a comic-specific notion of panels. While generic approaches could achieve similar ends, they would not match the expressivity found by localizing the syntax to the domain. **ComicScript** has limited support for data exploration but enables interactive data comics in a way that would otherwise be unmanageable [81]. As with DSLs in any domain, this may come at the cost of generality.

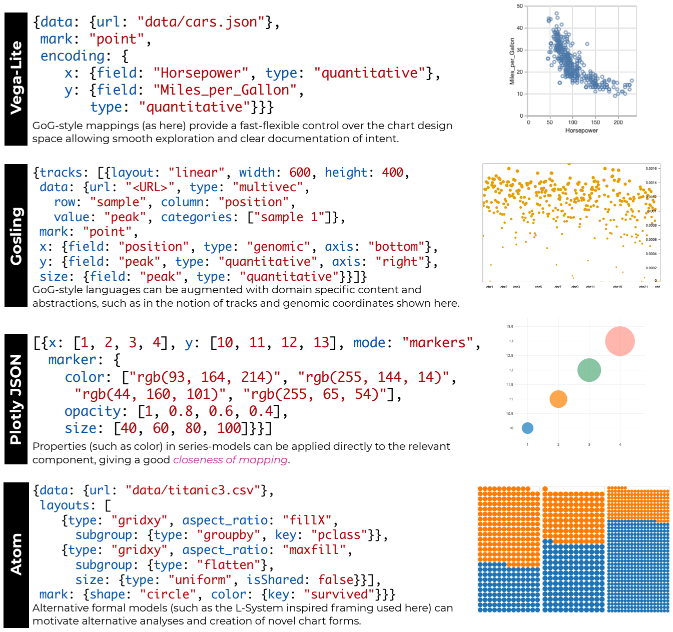
Surfacing domain-specific concepts allows the user to avoid the Turing tarpit and directly address aspects relevant to the domain. However, JSON-style DSLs are most useful for tasks that specifically fit the medium or that an end-user would wish to accomplish as a DSL (or through a facade for one). For instance, several virtual or extended reality systems (XR)—such as **DXR**, **VR-Viz**—use JSON-based grammars as the basis of their syntax. We suggest the prevalence of this approach may be because of XR's need to pass between mediums (e.g. between JS and Unity) is well matched with the portability of JSON-style DSLs. Tasks with less structure may be better matched with free-text DSLs (e.g. ViSlang [65]) as these allow for greater flexibility at the expense of automation (e.g. GUIs or recommenders).

4.2 Models: What is it?

All languages are predicated on a model of what computation is executed based on the commands written by the programmer. DSLs in our survey used models that fell along axes of low to high abstraction, and formal to colloquial. Drawing on prior work [84], abstraction level refers to DSLs that are close to the data domain as "high-level" (e.g. **VizML** or **Highcharts**) and those near the rendering context as "low-level" (e.g. **Vega** or **deck.gl/json**). Model formality denotes a pervasive logical structure in the DSL's design.

Formal Models. Languages backed by overarching frameworks are an important approach to visualization system design and are successful in tools like Tableau and ggplot. These *formal* models can simplify the expression of intent (within their scope) and aid potentially difficult analyses, although possibly impeding flexibility. We observed a variety of models whose purpose and intent varied.

The most prominent of these is Wilkinson's GoG [83] model in which data attributes are mapped to encoding channels and combined through marks. These forms allow for expressive construction and combination of visualizations allowing for the fluid creation of novel forms without concern for chart type. Examples of this form include **Vega-Lite**, **Vega**, **VizML**, **JSOL**, and **Flex-ER**. We suggest that this form is a safe default model for many visualization systems, as it "expose(s) the mechanics of good practice" [23]. Wilkinson has argued [63, 83] that his Grammar of Graphics is the *only* grammar of graphics. While this framing has been enormously successful in the development of visualization systems and languages, it is far from the only conceivable systematic model for creating visualizations. Others can be more tightly tuned to support particular tasks. For instance, **VizQL** emphasizes data exploration through the language of a data cube.



```
{data: {url: "data/cars.json"},
 mark: "point",
 encoding: {
    x: {field: "Horsepower", type: "quantitative"},
    y: {field: "Miles_per_Gallon",
        type: "quantitative"}}}
```

GoG-style mappings (as here) provide a fast-flexible control over the chart design space allowing smooth exploration and clear documentation of intent.

```
{tracks: [{layout: "linear", width: 600, height: 400,
 data: {url: "<URL>", type: "multivec",
   row: "sample", column: "position",
   value: "peak", categories: ["sample 1"]},
 mark: "point",
 x: {field: "position", type: "genomic", axis: "bottom"},
 y: {field: "peak", type: "quantitative", axis: "right"},
 size: {field: "peak", type: "quantitative"}}]}
```

GoG-style languages can be augmented with domain specific content and abstractions, such as in the notion of tracks and genomic coordinates shown here.

```
[{x: [1, 2, 3, 4], y: [10, 11, 12, 13], mode: "markers",
  marker: {
    color: ["rgb(93, 164, 214)", "rgb(255, 144, 14)",
      "rgb(44, 160, 101)", "rgb(255, 65, 54)"],
    opacity: [1, 0.8, 0.6, 0.4],
    size: [40, 60, 80, 100]}}]
```

Properties (such as color) in series-models can be applied directly to the relevant component, giving a good *closeness of mapping*.

```
{data: {url: "data/titanic3.csv"},
 layouts: [
    {type: "gridxy", aspect_ratio: "fillX",
      subgroup: {type: "groupby", key: "pclass"}},
    {type: "gridxy", aspect_ratio: "maxfill",
      subgroup: {type: "flatten"},
      size: {type: "uniform", isShared: false}}],
 mark: {shape: "circle", color: {key: "survived"}}}
```

Alternative formal models (such as the L-System inspired framing used here) can motivate alternative analyses and creation of novel chart forms.

Fig. 5. A DSL's domain and model manifest themselves in its syntax.

While many DSLs expressively use a declarative mapping of data to visual properties, other formulations can also be effective. For instance, **Atom** uses an L-system-inspired model to describe unit visualizations (see Fig. 5), whose graphical forms carry pivot sequences that are well matched with the iterative stages of an L-system. **CompassQL** and **SetCoLa** use constraints to generate programs that can act as a facade for complex systems whose interface might not be easy to understand or write (such as recommendation systems). **Scholz 3D Vis Language**, **Shih Volume Vis Language**, and **P4-6** use a sequential pipeline model. Pipeline models seem to be especially suited to low-level graphics tasks (as in **Shih Volume Vis Language** or shaders) or data manipulation (as in **P5** or **Vega**'s data model) as they may involve iterated stages whose results must be produced sequentially. We highlight these alternative framings because they may yield new approaches to various problems and enable chart types or analyses which are impossible (or needlessly difficult) in other framings. Alternative conceptual models are not unique to our survey. For instance, the HiVE notation [73] eases exploration of hierarchical orderings in treemaps. The development of DSLs for particular chart types or aspects is an intriguing opportunity for this genre of work, as are DSLs that surface novel semantic models as a way to expose new framings for analysis.

While our survey criterion requires that every DSL have algebraic qualities, several DSLs introduced explicit algebraic structures. Both **VizML** and **VizQL** use table algebras to describe the way data is manipulated prior to graphical display. **Vega-Lite** uses a simple data table model but provides a view algebra that provides affordances for viewing permutations and combinations. These algebras allow for rich expression within their domains. Similar systematic models might be established to serve other tasks or simply to surface alternate analysis approaches. For instance, **NEO** uses an algebra specific to confusion matrices. The visualization process has many interrelated steps, many of which might be enriched through formal modeling.

Colloquial Models. The complement to formal models are those models that do not impose a framework over the structure of the interface. These *colloquial models* are sometimes overlooked despite their ability to adapt and accommodate real-world situations and problems.

These less structured DSLs typically utilize series-based models in which data is mapped to aspects of *particular* chart forms (as in templates), such as the x-axis of a scatterplot or the angles of a pie chart (e.g. **Plotly JSON** in Fig. 5). Systems whose specification unit is a layer (such as **deck.gl/json** and **Bertin**) can be seen as using a series model in which the series are superimposed. They can be used at any level of abstraction, such as high-level DSLs like **AniVis** (which focuses on high-level chart templates) or low-level DSLs as in **deck.gl/json** (which can give access to shader-level manipulations).

Highcharts

```
{series: [{type: "treemap", data: […], layoutAlgorithm: "stripes",
  alternateStartingDirection: true,
  levels: […, {level: 1, layoutAlgorithm: "sliceAndDice",
    dataLabels: {enabled: true, align: "left"}}, …]}]}
```

Fig. 6. Colloquial DSLs can enable specification of chart attributes without modeling those attributes through the entire system.

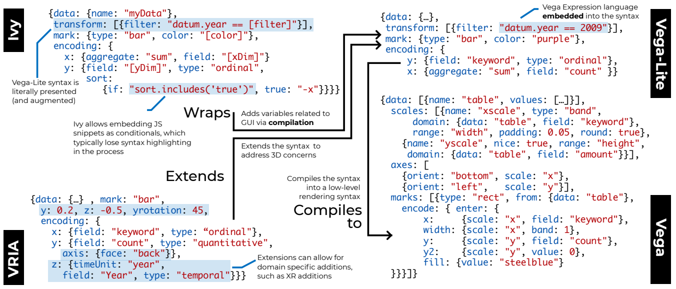
While these approaches are sometimes denigrated for their perceived lack of expressivity [18, 83], the close connection between their inputs and outputs forms a *closeness of mapping* that can make them simple to understand and easy to verbally describe (aspects which formal approaches may fall short on). This closeness can make the process of switching to a conceptually related (but visually distinct) chart form more *viscous*, as the two syntaxes may be highly different. They are more likely to allow unusual graphics as they can be created without respect to a formal model, such as **Bertin**'s multiple forms of cartograms or **ZingChart**'s funnel charts. These interfaces can be tuned to specifically support the domain, as in **Highcharts**'s treemaps (Fig. 6). However, their design is often ad hoc and may be *inconsistent* with other parts of the system, rendering them harder to learn or understand. This approach allows for simple local styles to be applied without needing to map those properties through top-level language concerns, as well as hierarchical styling which can be difficult to apply in systems whose data model requires tabular normalization for rendering (as in **Vega**).

Colloquial models can support particular features without forming themselves around that concept. For instance, only a handful of DSLs have top-level annotation support (e.g. **Plotly JSON**, **Highcharts**, and **ApexCharts**), however most of these are industry-driven efforts that do not use a formal model—**P4** and **Cicero** excepted, as they explicitly model annotation. Similarly, only a small set of mostly industry-led DSLs provide their own accessibility features (**Highcharts**, **Vega-Lite**, **ECharts**, and **FusionCharts**). This dearth may be due to the fact that such practicalities are often viewed as mere implementation details rather than being central to usability [57, 76], which may be compounded by a lack of research incentives to provide usable artifacts.

Beyond explicitly modeling features, formal models can provide these model-breaking behaviors through *escape hatches* to non-declarative programming or other points of extension (Sec. 4.5). However, doing so requires that such hatches be pre-placed in a manner relevant to the new feature—which is difficult to predict. Exploration of DSL malleability in a manner that allows for undesigned features without becoming unapproachably complex for end-users (a concern for the deeply malleable Varv [2]) is a valuable opportunity.

Formal models may have colloquial components. These are often found at interfaces with other systems. Styling is one such common leaky abstraction. The manner in which a chart is rendered may leak into the formalism without being modeled. For example, **Vega-Lite**'s concept of styling is guided by its downstream SVG and canvas renderers. Colloquial DSL components are not inherently detrimental. If the leaked feature is large or complex, it may be appropriate to embed that language rather than model it, although this can lead to *inconsistent* interfaces that are hard to adapt to new forms. Consideration of these properties in DSL design may be valuable, such as by designing with leaking in mind or by separating external concerns into separate DSLs.

Despite their academic stigma, we suggest that colloquial models may be valuable to consider. Among series-based DSLs we observed 306 distinctly named series types, although there was significant overlap in this list due to synonyms or simple modifiers (e.g. "3D" or "draggable"). The design space of name and modifier-based specification appears to be a rich one, although an analysis of which is beyond the scope of this work. Developing a better understanding of these forms and the way they are used by system designers and domain experts is intriguing work—especially in light of the decades-long popularity of this specification style (cf. SpotFire, **ChartML**, **XML Charts**). As researchers and tool-smiths [5], we suggest that we should meet people where they are, which may mean working to enrich colloquial models.



Ivy
Vega-Lite syntax is literally presented (and augmented)
Ivy allows embedding JS snippets as conditionals, which typically lose syntax highlighting in the process
Wraps
Adds variables related to GUI via **compilation**
Extends the syntax to address 3D concerns
Extends
Compiles the syntax into a low-level rendering syntax
Compiles to
Extensions can allow for domain specific additions, such as XR additions
VRIA
Vega Expression language **embedded** into the syntax
Vega-Lite
Vega

```
{data: {name: "myData"},
 transform: [{filter: "datum.year == [filter]"}],
 mark: {type: "bar", color: "[color]"},
 encoding: {
   x: {aggregate: "sum", field: "[xDim]"}
   y: {field: "[yDim]", type: "ordinal",
     sort:
       {if: "sort.includes('true')", true: "-x"}}}}
```

```
{data: {…} , mark: "bar",
  y: 0.2, z: -0.5, yrotation: 45,
  encoding: {
    x: {field: "keyword", type: "ordinal"},
    y: {field: "count", type: "quantitative",
      axis: {face: "back"}},
   z: {timeUnit: "year",
     field: "Year", type: "temporal"}}}
```

```
{data: {…},
 transform: [{filter: "datum.year == 2009"}],
 mark: {type: "bar", color: "purple"},
 encoding: {
     y: {field: "keyword", type: "ordinal"},
     x: {aggregate: "sum", field: "count" }}
```

```
{data: [{name: "table", values: […]}],
  scales: [{name: "xscale", type: "band",
      domain: {data: "table", field: "keyword"},
      range: "width", padding: 0.05, round: true},
    {name: "yscale", nice: true, range: "height",
     domain: {data: "table", field: "amount"}}],
  axes: [
    {orient: "bottom", scale: "x"},
    {orient: "left",   scale: "y"}],
  marks: [{type: "rect", from: {data: "table"},
    encode: { enter: {
        x:     {scale: "x", field: "keyword"},
        width: {scale: "x", band: 1},
        y:     {scale: "y", field: "count"},
        y2:    {scale: "y", value: 0},
        fill: {value: "steelblue"}
  }}}]}
```

Fig. 7. Annotated examples of the relationships between several DSLs.

4.3 Relationships: How does it relate to other DSLs?

Languages rarely exist in a language vacuum. For instance, many languages support regular expressions in a syntax unrelated to their own. We found that our DSLs are related to each other by Compiling, Wrapping/Embedding, or Extending/Contracting. The way a language relates to others determines numerous details about its implementation (particularly its execution model), although each relation has tradeoffs—the main tension residing between language customization and reuse.

Language composition can allow DSLs to be developed without requiring re-implementation or invention of abstractions. Yet, this comes at a cost as the new system is limited by the design choices of the old. For instance, compiling into **Vega-Lite** ensures that **Cicero** does not require the design of a visual encoding system, however, doing so means that it is restrained to the mark types available in **Vega-Lite** (and by a similar composition, **Vega**); impeding forms such as cartograms or treemaps. While such tradeoffs can be navigated, compositions that surface nested DSLs to the end user, as in **Scholz 3D Vis Language** use of **Vega** specifications, may diminish the value of a language style API, As they may necessitate frequent reference to documentation of the nested-DSLs rather than providing a single coherent expression language—yielding *language cacophony*.

Compile. Among *external* DSLs, there are two evaluation mechanisms: compilation and interpretation [17]. We refer to compilation as a process that generates code, while interpretation evaluates it directly. *Internal* languages are embedded into their host which carries with them all of the benefits and drawbacks associated with the more generic DSL design decision of internal vs external.

Compiled languages allow the language to gain all the strengths—and weaknesses—of its compile-target (often an interpreted or internal DSL), such as with **Cicero**. Compilation does not require a direct translation but can be used to embed computations into the resulting system. **SetCoLa** uses this strategy to create circular layouts which are not present in its compile-target WebCoLa [14]. In addition to offloading rendering, this gives access to potentially hard-to-achieve functionality, such as accessibility features.

Compilations may be chained together in a *compile tower* in a manner supported by targeted languages that do one thing well (akin to the Unix credo). For instance, our Gantt chart example in Fig. 4 might usefully target **Vega-Lite** to gain features such as tooltips without requiring implementation of nuanced details. As the ecosystem of JSON-style DSLs continues to grow it may be advantageous to select designs that re-use as much work of the previous DSLs as possible.

These benefits do not come for free. While **Vega-Lite** provides ARIA-accessibility features, it currently cannot provide some accessibility-enhancing encodings (e.g. texture) because its compile target, **Vega**, does not support them. More generally, errors may be harder for the end-user to understand if they are generated by the target's interpreter, whose concerns and conceptual model may be different from the source language. **Ivy**, which is a wrapping language that *uses* compilation, exemplifies this duality. It is language-agnostic and can be used over any JSON language, however, doing so precludes the surrounding application from providing contextual hints because it is unaware of the languages over which it is executed. Despite this, we argue that while compile towers are not always applicable, they should be employed more often. They can simplify implementation, improve usability, and reduce reinvention. They are well suited to research systems (whose contribution is not based on implementation) as they can rely on another system for repeatability and defense against bit-rot.

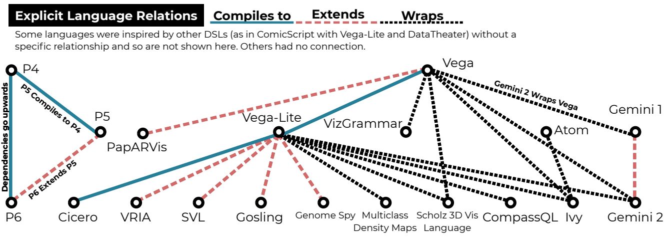


Fig. 8. DSLs can hold a variety of relationships with one another that allow them to reuse implementations, syntax, or concepts.

In contrast, interpretation allows for rich customization that can be helpful in specialized contexts. **Kyrix-S** supports large data sets for zooming visualizations. **Data Theater** uses an unusual data model (the output of an end-user specified Python script) to create explorable explanations. This approach can enable construction of contextual error messages (and other usability features) that are relevant to the local domain as they are not predicated on layers of indirection. However, this approach pushes rendering, data manipulation, and usability features onto the language implementer. Medium-focused DSLs tend to use this approach, possibly because their value is related to their customization to that medium (the main exceptions to this are XR-focused DSLs).

Wrap/Embed. Wrapping or embedding languages provides functionality extensions by literally containing other languages. For instance, **Gemini 2**, allows users to describe keyframes of an animation by explicitly including **Vega** and **Vega-Lite** specifications. **Scholz 3D Vis Language** allows the inclusion of entire **Vega** and **Vega-Lite** charts in a 3D context. There is overlap with compilation (as it can be used as a wrapping mechanism, as with **Ivy**), however we delineate this as a separate pattern to highlight the particular form of re-use. This approach allows for language-level separation of concerns as well as the use of the imported DSL's externalities (e.g. documentation or community support). This approach's main risk (beyond language cacophony) is that the embedded language might not match the domain and lead to *inconsistencies*.

A less extreme example of this approach is to embed language snippets—such as in the manner that SQL snippets are represented as strings in GPLs. These snippets address common tasks, such as control-flow or formatting (often via the d3-format language) as well as model-specific issues. **Vega** has a purpose-built JS subset for interacting with event streams. **FusionCharts** permits HTML snippets in tooltips. This common DSL pattern [17] allows for rich expression of intent, but may come at the cost of tooling, yielding some usability features (e.g. syntax highlighting) unavailable. There may be *hidden dependencies* within the snippet, as in the often numerous signals in **Vega** expressions [25].

Extend/Contract. An associated relationship is extension, in which a DSL is contracted or extended to form a new DSL. This usually comes in conjunction with syntactic extensions or modifications to the execution strategy. **Genome Spy** and **VRIA** contract the syntax of **Vega-Lite** behind custom renderers, and extend it with some genome and XR-specific affordances, respectively. **PapARVis** wraps and extends **Vega** with augmented reality enhancements. This approach can be useful as it allows for porting of ideas to new domains (e.g. **Genome Spy**'s reuse of **Vega-Lite** syntax in genomics). However, it does so at the expense of creating a new backend for that system. Given the variety of functionality developed across these DSLs, enabling their composition to allow greater reuse and increase their long-term impact.

Language elements are sometimes extended or reused in an ad hoc manner, a pattern which is more closely aligned with influence than extension. Some languages (such as **Encodable**, **Flex-ER**, or **DXR**) explicitly mold themselves on the thin mapping style of **Vega-Lite** or **Vega** without actually reusing the specific syntax or rendering systems. Other DSLs include only minor syntactic elements, such as **P4** and **Ivy**'s use of MongoDB-style operators (e.g. `{field: {$not: {OPERATOR-EXPRESSION}}}`). At other times this influence is conceptual. **GoTree** includes a spacing system related to the CSS box model, while **ZingChart** and **Cicero** include responsive-design features inspired by CSS media queries. Familiar syntax and concepts may aid learnability—possibly reducing the negative effects of *language cacophony*—however not every domain will fit every imported idea. For instance, CSS-style declarative rules are unlikely to handle the iterative stages of a data transformation pipeline well.

4.4 Language Affordances: Who is the end-user?

Each language has at least a general idea of its users, which motivates what features to include. We saw three user types based on the way they are expected to use the DSL, whose interests are naturally in tension. Some users simply use the DSL (*end-users*), others can modify the system which houses the DSL (*system-builders*), while others automatically manipulate and analyze the DSL (*automated agents*).

Syntax. A common first choice is whether to create an internal or external DSL. This is covered at length in other venues [17,53] but in essence, it can be seen as a question of language invention or embedding. The languages in our survey do not demonstrate any substantial divergences from the common benefits and limitations of each of these patterns. External DSLs offer richer expressivity but can be harder to construct and learn. Internal DSLs are easier to use within a host language but can force a notation that is poorly matched with the domain.

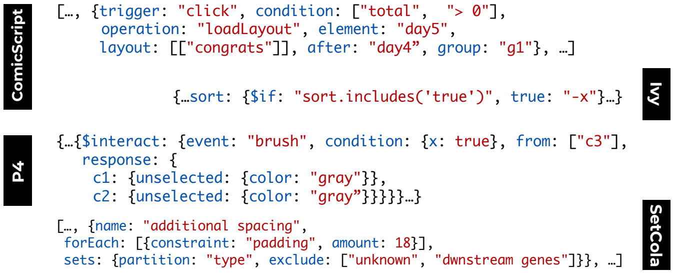
Internal languages seem to be most useful to interface-builders [84] as opposed to end-users. For instance, they do not by default facilitate automated analysis, and as such analysis can require dedicated high-complexity tools (such as AST-analyzers). These components are often beyond the design goals of tools meant only to support web-based presentations (e.g. **Chart.js**). Notebook-based analysis is a notable exception as it blurs end-user and system-builder. Internal bindings to external DSLs (e.g. Altair [78]) seem well matched to such hybrid users. However, such an analysis is beyond the scope of our study.

Among external languages, JSON is sometimes chosen for being end-user friendly. Scholz [70] notes that JSON was selected because it is human-readable and easy to transfer on the web. DeLine valued "YAML's declarative, hierarchical syntax" [10]. In contrast, a common criticism of XML as a carrier language is that it is verbose, which is seen as poorly matched with human usage [17, 83]. JSON's syntax, which is *terse*, appears to overcome this hurdle and may account for this style of DSL's growing popularity (Fig. 2).

Several end-user-focused systems expose their syntax to the end-user in applications. **StructGraphics** uses a custom GUI, whereas others (e.g. **Ivy** and **Glinda**) use plain text extended by modern editor affordances (e.g. autocomplete). Other DSLs have editors that support their use (without being required), such as **Gemini 1-2** or **Vega**. Constructing an environment around a DSL allows debugging tools and other end-user supportive features, however, this can (and has [90]) led to a constellation of small applications that repeatedly reimplement similar functionality. We suggest that it may be beneficial to consider how these efforts may be consolidated for this language style more generically.

Shih et al.'s [72] rationale for selecting JSON for their scientific visualization grammar was more focused on machine usability, noting that they selected "JSON because it is a widely used standard, is easy to parse, and it has sufficient expressiveness for hierarchical structures", an attitude shared in the design of **GoTree** [41]. Wu et al. [89] note that this interface style allows for manipulation by humans and autonomous users. While true for any executable language, manipulations are easier in serialization formats due to their restricted form. The limited grammar allows for exhaustive design space exploration, enabling recommendation (as in **CompassQL** and **Cicero** [32]) and enumeration of novel chart forms (as in **Atom** [57] and **GoTree** [41]).

An often discussed benefit of external DSLs [53] is that they expose a notation local to a domain—as in Diderot's [7] explicit use of tensor operators (e.g. ∇ and ⊛). While JSON-like languages can abstract over various domains, few domains use it as their primary notation (API design and data definition are clear exceptions). The selection of these carrier languages as syntax is then a compromise. In exchange for benefits like portability and simple machine operability, domain experts encounter a less familiar notation.


ComicScript
Ivy
P4
SetCola

```
[…, {trigger: "click", condition: ["total",  "> 0"],
      operation: "loadLayout", element: "day5",
      layout: [["congrats"]], after: "day4”, group: "g1"}, …]

                    {…sort: {$if: "sort.includes('true')", true: "-x"}…}

{…{$interact: {event: "brush", condition: {x: true}, from: ["c3"],
   response: {
    c1: {unselected: {color: "gray"}},
    c2: {unselected: {color: "gray”}}}}}…}

[…, {name: "additional spacing",
 forEach: [{constraint: "padding", amount: 18}],
 sets: {partition: "type", exclude: ["unknown", "dwnstream genes"]}}, …]
```

Fig. 9. Many languages feature logic or control flow operators.

Abstraction Mechanisms. Creating *abstractions* is an important part of any programming language. In GPLs features like variables, functions, if-else structures, loops, and a host of others serve this purpose. Some languages in our survey utilized these elements (Fig. 8) allowing abstraction on syntactic, data, output, or contextual levels. Those that did not, likely did so to limit scope, because their domain did not require it, or relied on their host for such features (a benefit of internal DSLs). The tendency to forgo abstraction in DSLs is well known [17], but we highlight it to explore the particularities exemplified in this context.

Control flow operators (as in Fig. 9) were common. These conditionals can address a range of program aspects including the data (as in **Vega-Lite**'s conditional marks), the graphic (as in the query selectors found in **Canis**), interactions (as in **ComicScript**), or container state (such as in languages rooted in GUIs like **Ivy**). Some focused on modifying graphics based on interactions (as in **P4**) while others focused on syntactic transformations (as in **Ivy**). Some languages, such as **Vega**, use another language to evaluate their condition (via embedded snippets), while others, like **ComicScript**, construct the logic through explicit operators. While it is not necessary to be able to query or conditionalize every element, each of them can be beneficial depending on the domain although not every situation necessitates such facilities. **Cicero** uses a powerful query language that gives access to data, graphic, and specification, which is necessary for its responsive and annotation tasks, although it appropriately has no concept of its surrounding context (besides aspect ratio). Embedding snippets offers greater expressivity [17] (potentially at the price of portability and diminished usability features), while explicit operators allow the user and their tooling to keep a single consistent mental model (potentially introducing unfamiliar syntax).

A variety of other abstraction mechanisms were used. Some systems included notions of variables, although their purpose varied. **Flex-ER** and **Vega** use FRP-style signals as variables. **Canis** uses variables as a form of textual-macro replacement. **Ivy** and **Vega-Lite** use variables as a way to reference GUI controls exposed to the end-user (although **Vega-Lite**'s are a mask for **Vega**'s signals). Variables can help reduce the cognitive load on the user by reducing *diffuseness*, but it can also increase it if the references become difficult to follow. **SetCoLa** was the only language to include loops. While an appropriate syntactic choice for their domain (simplifying constraint generation) it is common for DSLs to not provide loops as this can cause DSLs to accidentally "slide into generality" [17]. None of the *external* DSLs had SQL-style end-user definable functions. Varv [2] takes extensibility to an extreme via a fully end-user editable application creation external DSL (that includes simple macros); demonstrating that this level of malleability is achievable in external DSLs.

The selected *abstraction gradient* should cater to a designed audience. Loops and variables can help readability, which supports humans but does not generally affect automations. Functions and control flow operators can aid in reuse, but if programs are generated on the fly in a GUI and not meant for reuse, their utility will be limited. If the intended user's interests are not aligned with multiple such user types then a different interface may be preferential to a JSON-style DSL. For instance, a DSL solely focused on humans using notebooks will likely be better served by not imposing the grammatical limitations of a serialization language, while an automatically generated language for facilitating chart recommendations need not be human readable.

4.5 Practicalities: Where is work done?

There are a number of places within a DSL where a given feature can be implemented. As in Fig. 10, these include explicit and implicit modeling as well as internal and external placement. This modeling describes where the user is expected to do work to use those features.

Each strategy has advantages and disadvantages. Internal features give deep control over implementation. However, that entity must be clearly represented or risk *visibility* errors. Explicit modeling can allow the user to address a task directly, but it can require the development of new (potentially *inconsistent*) syntax. External features can push burdens to other systems, reducing portability and potentially inducing *hidden dependencies*. Implicit modeling can be deeply expressive, but carrying out such intentions can yield *hard mental operations*.

Integration and State. Most DSLs manage state and interactivity through a runtime inside the system. This allows them exact control over the way a feature is delivered, and is thus favored by internal and interpreted DSLs. Similarly, internal DSLs allow rich integration with web pages through affordances like callbacks—typically in exchange for a lack of end-user control. Other systems manage state by integration with an external application. **StructGraphics**, for instance, provides a visualization builder interface that maps spreadsheet data to graphics. Embedding state into a housing application allows graphics to be synchronized with and used to control the UI, enabling deeply integrated experiences, although this may impede portability. Some systems (typically compiled DSLs) pass control to an external system, as in **Vega-Lite** or **SetCoLa**. This simplifies system construction, however, it may impede interactions outside the target's model.

These approaches only have value when appropriately coordinated with their purpose. For instance, **Atom** uses a custom interpreter integrated into an application. While this approach allows **Atom** to be closely integrated with its editing environment, it makes it difficult to *portably* reuse specifications in other contexts and precludes them from being integrated into other applications. Given its position as an academic artifact whose value is not related to its renderer, we suggest that DSLs like **Atom** may be well matched with a *compile tower*-style strategy to alleviate implementation burden and facilitate portability.

Some DSLs provide mechanisms for interactions and integration with their environment, although this can require coupling with those systems (as does binding to any external system). For instance, **Encodable** explores creating **Vega-Lite**-style facades over arbitrary JS visualizations. However, this causes those little languages to be inextricably linked to JS. We suggest that *portability and contextual integration are opposing goals*, as surfacing integrations as first-class aspects of the language creates a context dependence. Both are reasonable design choices, but favoring integration may reduce some of JSON-style DSLs' utility (e.g. portability). Yet, language-level integration in these DSLs is unexplored, so it may be useful to consider visualizations as part of a system rather than singular units.

Alternate language APIs. Some DSLs are used in host languages through internal bindings. This type of tool can enable work-arounds for DSL limitations by externalizing these needs to a host language, as in the Gos Python-wrapper for **Gosling** [45] or Altair [78] for **Vega-Lite**. For instance, some facet and layer combinations create data ambiguities that can prevent **Vega-Lite** from rendering. This can be resolved by manually pivoting data and combining Altair charts in Python.

Beyond providing workarounds for language issues, this can simplify program specification for users with limited familiarity with the DSL. For instance, it may be easier for an Elm programmer to use elm-vega [87] than to a write a corresponding **Vega** specification. This is analogous to the value of object relational mappers for manipulation of SQL databases. Users can write and reason about their database in their chosen language rather than being required to utilize SQL's sometimes idiosyncratic or unfamiliar form. We emphasize that consideration of the environments in which a language will be used is a valuable component of language design as it may surface components and strategies that guide the design, such as expecting faceting to be done outside of the DSL.

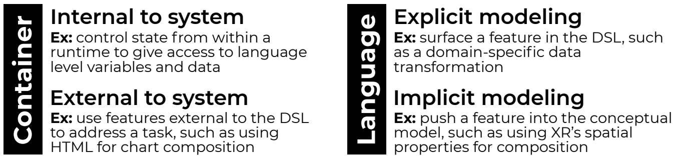

Container
Internal to system
Ex: control state from within a runtime to give access to language level variables and data
External to system
Ex: use features external to the DSL to address a task, such as using HTML for chart composition
Language
Explicit modeling
Ex: surface a feature in the DSL, such as a domain-specific data transformation
Implicit modeling
Ex: push a feature into the conceptual model, such as using XR's spatial properties for composition

Fig. 10. Features can be built in a variety of places across the DSL.

Extensibility. Wilkinson notes that any closed system will have missing pieces [83]. For instance, his **VizML** has limited support for nested or data-driven layouts (e.g. sets or cartograms) or mixed data and aesthetic-driven tasks (such as annotation). Some DSLs approach extensibility by designing places where external elements can be introduced into the system through an API. This can allow for the introduction of new transforms (as in **Vega-Lite**), user-defined marks (as in **DXR**), renderers (as in **Vega**), as well as chart types or events (as in **Chart.js**). Extensions typically occur in explicitly modeled features within the system, precluding end-user modification of system concepts (e.g. new coordinate systems). Open source software can allow for a slower but less limited form of extension. Yet this is not always the case. Some systems are no longer maintained, might be resistant to external changes, or might require too high a technical barrier to contribution from domain experts.

A form of extension available in some formal model-based DSLs is the creation of ad hoc mark types within the language itself. Wongsuphasawat [85] explores how end-users of some systems can construct composite marks, such as candlesticks, although there are boundaries to this imposed by the form of the language model. For instance, **Vega-Lite** allows some custom glyph creation (via image marks), which enables unit isotypes but not aggregates—as such encodings fall outside its model. Some extensions are not possible without external modeling, as in our Gantt example (Fig. 4). This extension style is powerful but is limited by its model and so can be well paired with external extension.

Combination and Data Strategy. A DSL's approach to image composition (e.g. layering or juxtaposition) and handling data exemplify the stratification of *where* work is done.

We observed a spectrum of strategies for image composition ranging from specification *above* the language (in the container), explicitly modeling *within* the language, to *below* it implicitly in the conceptual model. Many DSLs do not provide a mechanism for combination, either because it is not relevant to their domain (as in graph DSLs) or by making use of awareness of their medium as the implied context through which conjunction happens. For instance, **ECharts** relies on the browser for spatial arrangement and the user for data partitioning. This can be simple to construct but pushes implementation onto the user.

Some formal models feature a composition algebra, as in **Vega-Lite**'s layer, facet, and concat operations. These operators typically focus on data partitions (to facilitate small multiples), however, Wu also describes an under-explored notion of parameter-based faceting [90]. Some languages include a combination mechanism unique to their domain, such as **ComicScript**'s panels or **Gosling**'s notion of tracks. These approaches are useful, but typically require explicit modeling in the language, which can take up limited conceptual real estate.

Some DSLs push feature description into their conceptual model, such as by using spatial position for combination (as in map and XR DSLs). While powerful, these should be used cautiously as implicit operations can yield *hard mental operations*.

The selection of how and where data is handled is critical, as it determines how a DSL can interact with its environment and compose with other DSLs. Most DSLs hold all their data inside the system, a simplistic model which is adequate for many use cases, however some externalize that task. For instance, **Kyrix-S** uses a custom back-end that sits on top of a database to allow exploration of large datasets via zooming. While sometimes useful, a complex data strategy is unlikely to work with a system that does not share that strategy: **Kyrix-S** is unlikely to be interoperable with **Multiclass-Density-Maps** despite sharing a domain interest in aggregating heatmaps. Similarly, DSLs exhibit no manipulation strategy (ignoring it or externalizing to the host), rudimentary language manipulations such as filters, or a richer domain-relevant expression or transformation system explicitly modeled within the language (as in **Genome Spy**'s genome-focused transformations). These carry similar tradeoffs as composition in placement stratification. Unlike composition, there are well-implemented libraries that support this task. Yet, many DSLs implement their data processing features themselves [90]. These recreations may be motivated by domain. For example, most data libraries do not support the genomic coordinates required by **Gosling**—although this is more often done needlessly when more robust implementations are available. Consideration of how and where to place features such as these is critical for DSL usability.

5 Discussion

In this study, we surveyed JSON-style DSLs for visualization from across academia, industry, and open source efforts spanning a period of more than 20 years. In doing so we examined both the state of the art for this domain (such as the role of DSL compilation vs interpretation or abstraction in DSLs), and introduced new concerns (including the tension between colloquial and formal visualization models), patterns (like the role of composition), and practices (e.g. supporting both computational and human users). We observed a wide variety of tasks and domains that this style of language seeks to serve, indicating its pliability to a large collection of concerns and highlighted many avenues for future work (such as designing languages that can be bidirectionally updated). From our results, we are optimistic about this style's future. However this is not without caveats nor enticing avenues of exploration.

Study Limitations. We sought to understand the design and implementation of JSON-style DSLs through analysis of artifacts, documentation, and scholarly works. While this revealed a number of intriguing patterns, it did not capture the entirety of visualization DSLs or APIs. For instance, we excluded a variety of visualization languages (e.g. ggplot) and JSON-based languages outside of visualization (e.g. Varv). Exploration of design patterns and tradeoffs found in these and other languages, libraries, and APIs is warranted in future studies.

Computer languages, like spoken languages, are often living entities whose design changes over time and can be driven by individuals' undocumented ideas or influences. In future work, this analysis might be enriched through interviews with DSL authors to better understand language design choices and life cycles.

Our survey was biased in several ways. Our survey was biased towards more recent open source and academic works, and away from older or privately-produced DSLs, as it is easier to find public contemporary systems. As such, there are likely additional DSLs that were not observed during our search. While additional data would be useful, we believe our sample sufficiently captures the tendencies of this language form, although sample size and biases are known issues [92]. Some codings were based on limited documentation as we were unable to locate some DSL artifacts (e.g. due to URL-rot or closed-source). We intend to continue expanding the corpus of examples in our supplement to facilitate further empirical investigation of this language genre. This may reveal patterns hidden from our qualitative lens. Finally, our analysis is limited by our own biases, which we sought to reduce through iterative theming and reflection.

Language Design and Tooling. The design of an effective DSL for achieving any of the nuanced tasks that visualization DSLs seek to solve is a thorny problem. Novel languages and notations have the potential to serve as foundations on which to "think the unthinkable" [80], but also can add needless complexity and cacophony. How to build powerful tools that do not result in confusion which can be applied to an ecosystem that has a competing set of users? Per the tensions and tradeoffs we highlight throughout this study, there is no one answer. We suggest that developing mechanisms for design evaluation and improving the DSL tool ecosystem may fruitfully guide future DSLs.

DSL evaluation is a long-running topic [3, 61]. However, these methods are not extensively used for visualization DSLs [63]. Only two works [11,69] offered a formal CDN analysis, while others provided ad hoc reasoning [83]. Poltronieri et al. [61] suggest that DSL evaluation may be more effective if done in a contextual and non-generic manner (as in Jakubovic et al.'s [29] work on programming systems or Elavsky et al.'s work on visualization accessibility [15]). Such a call might be answered with evaluatory heuristics, like *"What tasks does this DSL address?"*, *"What form of model is it using?"*, *"How are non-data elements described?"*, *"Who is the intended user?"*, or *"What is meant by data?"* A notable experiment in this regard is Pu et al.'s [62] use of Algebraic Visualization [34, 49] as a sibling to CDN, suggesting the applicability of visualization theory to DSL evaluation.

JSON DSLs have been described as being intended for use by end-users [50]. However, there has been little formal usability analysis. Hoffswell et al. [25] studied debugging in **Vega**. Naimipour et al. [54] explore social science teachers' perceived usability of **Vega-Lite**. These works demonstrate this approach's utility. However, future work should investigate which language form is best matched with end-users.

We believe that some of the usability issues found in JSON-style visualization DSLs [44] can be addressed through careful enhancements to end-user tooling. Merino et al. explore this in their system for creating notebooks tuned to individual DSLs [52]. Hoffswell et al. [25] augment textual representations of **Vega** programs with in-situ state visualizations. As JSON-style DSLs continue to be developed, it may be useful to explore *language workbenches* [17], which are a form of tool for designing, composing, and using (often domain-specific) languages. Some work has been done in this direction by JSON Schema structure editors [2]. However, they focus on data validation and not language design. Some DSLs provide formal syntax definitions. However, none formalize their *semantics*, although this may be because a visualization semantics language does not exist. We suggest that a metalanguage for such descriptions would be valuable future work.

JSON-style DSLs can be *error prone* through silent errors or overrides, such as those caused by invalid or misspelled properties. This can be confusing to the end-user who then receives little feedback on why execution is not carried out as they expect. Tools like JSON Schema can be useful to reduce this type of error, but they are only able to capture syntactic errors. Analysis tools like linters [6, 27, 51] can help capture semantic errors, although the configuration of which may present non-trivial complexities. Future designs should explore encoding invariants as syntax so that invalid expression is impossible.

The Next 10k Visualization Grammars. If our survey prompts any prediction, it is that *new visualization languages will continue to be developed*—some of which may be in the JSON-style. There are many forms, shapes, and purposes these languages may take. Hogräfer et al. [26] argue for a map grammar. Lau et al. [38] call for a computational notebook grammar that would enable task-specific notebook forms. Hullman and Gelman call for a grammar that enables statistical model checks [28]. Following the trend of developing DSLs to support complex data tasks in genomics [39, 45] or ML [43], other languages could be developed for other data-intensive contexts, such as multi-scale analysis, temporal data, or textual data. Tuning computation-heavy algorithms or processes involving randomness (as in force direction) can be clumsy and error-prone, suggesting that an end-user-centered DSL enhancing those operations might be valuable. The volume of XR DSLs suggests that JSON-style DSLs may be useful for other uncommon mediums. For instance, a sonification grammar (such as briefly explored by **Highcharts** and **DXR**) might make non-visual data experiences easier and more accessible to produce.

Park et al. [57] argue that efforts should be made to "find a definitive grammar that can unify many of these existing grammars". However, Greenspun's tenth rule [19] quips that any sufficiently complicated program contains an ad hoc, informally specified, bug-ridden, slow implementation of half of Lisp. Less satirically Fowler notes that one of the biggest dangers in DSL design is "Sliding into generality." [17] While tools like **Vega-Lite** are probably not in danger of becoming Lisp, we suggest that consideration of small modular language components may be helpful in the continuation and extension of this ecosystem.

There are many tasks, and no one DSL will be able to capture all of them without compromising essential parts of its domain design. That is, there is *no grammar to rule them all*.

Acknowledgments

We thank our reviewers, as well as Ravi Chugh, Arvind Satyanarayan, Brian Hempel, Will Brackenbury, and Michael A. McNutt.

References

[1] A. Blackwell and T. Green. Notational Systems–the Cognitive Dimensions of Notations Framework. *HCI Models, Theories, And Frameworks: Toward An Interdisciplinary Science. Morgan Kaufmann*, 2003.

[2] M. Borowski, L. Murray, J. B. Bagge, Rolf Kristensen, A. Satyanarayan, and C. N. Klokmose. Varv: Reprogrammable Interactive Software As a Declarative Data Structure. In *Conference on Human Factors in Computing Systems*, pp. 492:1–492:20, 2022. doi: 10.1145/3491102.3502064

[3] H. S. Borum, H. Niss, and P. Sestoft. On Designing Applied DSLs for Non-programming Experts in Evolving Domains. In *Conference on Model Driven Engineering Languages And Systems*, pp. 227–238, 2021.

[4] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[5] F. P. Brooks Jr. The Computer Scientist As Toolsmith II. *Communications of the ACM*, 39(3):61–68, 1996.

[6] Q. Chen, F. Sun, X. Xu, Z. Chen, J. Wang, and N. Cao. VizLinter: a Linter And Fixer Framework for Data Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2021.

[7] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: a Parallel DSL for Image Analysis And Visualization. In *Programming Language Design and Implementation*, pp. 111–120, 2012.

[8] K. Compton, B. Kybartas, and M. Mateas. Tracery: An Author-Focused Generative Text Tool. In *International Conference on Interactive Digital Storytelling*, pp. 154–161. Springer, 2015.

[9] M. Conlen and J. Heer. Idyll: a Markup Language for Authoring And Publishing Interactive Articles on the Web. In *Symposium on User Interface Software and Technology*, pp. 977–989, 2018.

[10] R. A. DeLine. Glinda: Supporting Data Science with Live Programming, GUIs And a Domain-specific Language. In *Conference on Human Factors in Computing Systems*, pp. 1–11, 2021.

[11] S. Devkota, M. Legendre, A. Kunen, P. Aschwanden, and K. E. Isaacs. CFGConf: Supporting High Level Requirements for Visualizing Control Flow Graphs. *arxiv*, 2021.

[12] Dhall. Design Choices. `https://docs.dhall-lang.org/discussions/Design-choices.html`, 2021. Viewed 1/3/21.

[13] T. Duplantis, I. Karth, M. Kreminski, A. M. Smith, and M. Mateas. A Genre-Specific Game Description Language for Game Boy RPGs. In *IEEE Conference on Games*, 2021.

[14] T. Dwyer. WebCola. `https://github.com/tgdwyer/WebCola`. Viewed 3/5/2022.

[15] F. Elavsky, C. Bennett, and D. Moritz. How Accessible Is My Visualization? Evaluating Visualization Accessibility with Chartability. In *Eurographics Conference on Visualization*, p. 14, 2022. To Appear.

[16] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language Composition Untangled. In *Workshop on Language Descriptions, Tools, And Applications*, pp. 1–8, 2012.

[17] M. Fowler. *Domain-specific Languages*. Pearson Education, 2010.

[18] M. Friendly. Colorless Green Graphs Sleep Furiously: a Conversation with Leland Wilkinson. *Nightingale*, March 2022.

[19] P. Greenspun. 10th Rule of Programming. `http://philip.greenspun.com/bboard/q-and-a-fetch-msg?msg_id=000tgU`, 2003. Viewed 1/3/2022.

[20] S. Günther, T. Cleenewerck, and V. Jonckers. Software Variability: the Design Space of Configuration Languages. In *Workshop on Variability Modeling of Software-Intensive Systems*, pp. 157–164, 2012.

[21] M. Guzdial and T. Shreiner. Integrating Computing Through Task-specific Programming for Disciplinary Relevance: Considerations And Examples. In *Computational Thinking in Education*, pp. 172–190. Routledge, 2021.

[22] B. Hasimoto. Glisp. `https://github.com/baku89/glisp`, 2021.

[23] K. Healy and J. Moody. Data Visualization in Sociology. *Annual Review of Sociology*, 40:105–128, 2014.

[24] B. Hempel, J. Lubin, and R. Chugh. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Symposium on User Interface Software and Technology*, pp. 281–292, 2019. doi: 10.1145/3332165.3347925

[25] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting Code with in Situ Visualizations To Aid Program Understanding. In *Conference on Human Factors in Computing Systems*, pp. 1–12, 2018.

[26] M. Hogräfer, M. Heitzler, and H.-J. Schulz. The State of the Art in Map-Like Visualization. In *Computer Graphics Forum*, vol. 39, pp. 647–674. Wiley Online Library, 2020.

[27] A. K. Hopkins, M. Correll, and A. Satyanarayan. VisuaLint: Sketchy in Situ Annotations of Chart Construction Errors. In *Computer Graphics Forum*, 2020. doi: 10.1111/cgf.13975

[28] J. Hullman and A. Gelman. Designing for Interactive Exploratory Data Analysis Requires Theories of Graphical Inference. *HDSR*, 2021.

[29] J. Jakubovic, J. Edwards, and T. Petricek. Technical Dimensions of Programming Systems. 2022.

[30] E. Jun, M. Daum, J. Roesch, S. Chasins, E. Berger, R. Just, and K. Reinecke. Tea: a High-level Language And Runtime System for Automating Statistical Analysis. In *Symposium on User Interface Software and Technology*, pp. 591–603, 2019.

[31] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. *arxiv*, 2014.

[32] H. Kim, R. Rossi, F. Du, E. Koh, S. Guo, J. Hullman, and J. Hoffswell. Cicero: a Declarative Grammar for Responsive Visualization. In *Conference on Human Factors in Computing Systems*, pp. 600:1–600:15, 2022. doi: 10.1145/3491102.3517455

[33] Y. Kim, K. Wongsuphasawat, J. Hullman, and J. Heer. GraphScape: a Model for Automated Reasoning About Visualization Similarity And Sequencing. In *Conference on Human Factors in Computing Systems*, pp. 2628–2638. ACM, 2017. doi: 10.1145/3025453.3025866

[34] G. Kindlmann and C. Scheidegger. An Algebraic Process for Visualization Design. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2181–2190, 2014. doi: 10.1109/TVCG.2014.2346325

[35] L. C. Klopfenstein, S. Delpriori, and A. Ricci. Adapting a Conversational Text Generator for Online Chatbot Messaging. In *International Conference on Internet Science*, pp. 87–99. Springer, 2018.

[36] A. Ko. Tweet, November 2021. `https://twitter.com/amyjko/status/1458537839939895299`.

[37] Y. S. Kristiansen and S. Bruckner. Visception: An Interactive Visual Framework for Nested Visualization Design. *Computers & Graphics*, 92:13–27, 2020.

[38] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia And Industry. In *Visual Languages And Human-Centric Computing*, pp. 1–11. IEEE, 2020.

[39] K. Lavikka. Grammar-Based Interactive Genome Visualization. Master's thesis, Helsingin yliopisto, 2020.

[40] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, and W. Chen. ECharts: a Declarative Framework for Rapid Construction of Web-based Visualization. *Visual Informatics*, 2(2):136–146, 2018.

[41] G. Li, M. Tian, Q. Xu, M. J. McGuffin, and X. Yuan. Gotree: a Grammar of Tree Visualizations. In *Conference on Human Factors in Computing Systems*, pp. 1–13, 2020.

[42] J. K. Li and K.-L. Ma. P4: Portable Parallel Processing Pipelines for Interactive Information Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 26(3):1548–1561, 2018.

[43] J. K. Li and K.-L. Ma. P6: a Declarative Language for Integrating Machine Learning in Visual Analytics. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):380–389, 2020.

[44] Z. Liu, C. Chen, F. Morales, and Y. Zhao. Atlas: Grammar-based Procedural Generation of Data Visualizations.

[45] S. L'Yi, Q. Wang, F. Lekschas, and N. Gehlenborg. Gosling: a Grammar-based Toolkit for Scalable And Interactive Genomics Data Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2022.

[46] J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *Acm Transactions on Graphics*, 5(2):110–141, 1986.

[47] N. Makrynioti and V. Vassalos. Declarative Data Analytics: a Survey. *IEEE Transactions on Knowledge And Data Engineering*, 2019.

[48] A. McNutt. On the Potential of Zines As a Medium for Visualization. In *IEEE Visualization Conference*, pp. 176–180. IEEE, 2021.

[49] A. McNutt. What Are Table Cartograms Good for Anyway? An Algebraic Analysis. In *Computer Graphics Forum*, vol. 40, pp. 61–73. Wiley Online Library, 2021.

[50] A. McNutt and R. Chugh. Integrated Visualization Editing Via Parameterized Declarative Templates. In *Conference on Human Factors in Computing Systems*, pp. 1–14, 2021.

[51] A. McNutt, G. Kindlmann, and M. Correll. Surfacing Visualization Mirages. *Conference on Human Factors in Computing Systems*, 2020. doi: 10.1145/3313831.3376420

[52] M. V. Merino, J. Vinju, and T. van der Storm. Bacatá: Notebooks for DSLs, Almost for Free. In *International Conference on Art, Science, And Engineering*. ACM, 2020.

[53] M. Mernik, J. Heering, and A. M. Sloane. When And How To Develop

Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[54] B. Naimipour, M. Guzdial, and T. Shreiner. Engaging Pre-service Teachers in Front-end Design: Developing Technology for a Social Studies Classroom. In *Frontiers in Education Conference*, pp. 1–9. IEEE, 2020.

[55] A. Narechania, A. Srinivasan, and J. Stasko. NL4DV: a Toolkit for Generating Analytic Specifications for Data Visualization From Natural Language Queries. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):369–379, 2020.

[56] D. Orchard. The Four Rs of Programming Language Design. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, And Reflections on Programming And Software*, pp. 157–162, 2011.

[57] D. Park, S. M. Drucker, R. Fernandez, and N. Elmqvist. Atom: a Grammar for Unit Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 24(12):3032–3043, 2017.

[58] W. C. Payne, Y. Bergner, M. E. West, C. Charp, R. B. B. Shapiro, D. A. Szafir, E. V. Taylor, and K. DesPortes. DanceON: Culturally Responsive Creative Computing. In *Conference on Human Factors in Computing Systems*, pp. 1–16, 2021.

[59] A. J. Perlis. Special Feature: Epigrams on Programming. *ACM Sigplan Notices*, 17(9):7–13, 1982.

[60] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *Conference on World Wide Web*, 2016. doi: 10.1145/2872427.2883029

[61] I. Poltronieri, A. C. Pedroso, A. F. Zorzo, M. Bernardino, and M. d. Borba Campos. Is Usability Evaluation of DSL Still a Trending Topic? In *International Conference on Human-Computer Interaction*, pp. 299–317. Springer, 2021.

[62] X. Pu and M. Kay. A Probabilistic Grammar of Graphics. In *Conference on Human Factors in Computing Systems*, 2020.

[63] X. Pu, M. Kay, S. M. Drucker, J. Heer, D. Moritz, and A. Satyanarayan. Special Interest Group on Visualization Grammars. In *Conference on Human Factors in Computing Systems*, pp. 1–3, 2021.

[64] X. Qin, Y. Luo, N. Tang, and G. Li. Making Data Visualization More Efficient And Effective: a Survey. *VLDB*, 29(1):93–117, 2020.

[65] P. Rautek, S. Bruckner, M. E. Gröller, and M. Hadwiger. ViSlang: a System for Interpreted Domain-specific Languages for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2388–2396, 2014.

[66] A. Satyanarayan, B. Lee, D. Ren, J. Heer, J. Stasko, J. Thompson, M. Brehmer, and Z. Liu. Critical Reflections on Visualization Authoring Systems. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):461–471, 2020. doi: 10.1109/TVCG.2019.2934281

[67] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: a Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2016.

[68] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: a Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2016. doi: 10.1109/TVCG.2015.2467091

[69] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative Interaction Design for Data Visualization. In *Symposium on User Interface Software and Technology*, pp. 669–678, 2014.

[70] D. Scholz. A Modular Domain-Specific Language for Interactive 3D Visualization. Master's thesis, TU Wien, May 2021.

[71] L. Shen, X. Chen, R. Liu, H. Wang, and G. Ji. Domain-Specific Language Techniques for Visual Computing: a Comprehensive Study. *Archives of Computational Methods in Engineering*, 28(4):3113–3134, 2021.

[72] M. Shih, C. Rozhon, and K.-L. Ma. A Declarative Grammar of Flexible Volume Visualization Pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1050–1059, 2018.

[73] A. Slingsby, J. Dykes, and J. Wood. Configuring Hierarchical Layouts To Address Research Questions. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):977–984, 2009.

[74] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages As Libraries. In *Programming Language Design and Implementation*, pp. 132–141, 2011.

[75] J. Tran O'Leary, K. Lee, and N. Peek. A Grammar of Digital Fabrication Machines. In *Conference on Human Factors in Computing Systems*, pp. 1–6, 2021.

[76] T. Tsandilas. StructGraphics: Flexible Visualization Design Through Data-Agnostic And Reusable Graphical Structures. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):315–325, 2021. doi: 10.1109/TVCG.2020.3030476

[77] A. Van Deursen, P. Klint, and J. Visser. Domain-specific Languages: An Annotated Bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[78] J. VanderPlas, B. E. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software*, 3(32):1057, 2018. doi: 10.21105/joss.01057

[79] B. Victor. Drawing Dynamic Visualizations, May 2013.

[80] B. Victor. Media for Thinking the Unthinkable. `http://worrydream.com/MediaForThinkingTheUnthinkable`, April 2013.

[81] Z. Wang, H. Romat, F. Chevalier, N. H. Riche, D. Murray-Rust, and B. Bach. Interactive Data Comics. *IEEE Transactions on Visualization and Computer Graphics*, 2022.

[82] H. Wickham. A Layered Grammar of Graphics. *Journal of Computational And Graphical Statistics*, 19(1):3–28, 2010.

[83] L. Wilkinson. The Grammar of Graphics. In *Handbook of Computational Statistics*, pp. 375–414. Springer, 2012.

[84] K. Wongsuphasawat. Encodable: Configurable Grammar for Visualization Components. In *IEEE Visualization Conference*, pp. 131–135. IEEE, 2020.

[85] K. Wongsuphasawat. Navigating the Wide World of Data Visualization Libraries. *Nightingale*, September 2020.

[86] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Conference on Human Factors in Computing Systems*, pp. 2648–2659. ACM, 2017. doi: 10.1145/3025453.3025768

[87] J. Wood, A. Kachkaev, and J. Dykes. Design Exposition with Literate Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):759–768, 2019. doi: 10.1109/TVCG.2018.2864836

[88] wso2. VizGrammar. `https://github.com/wso2/VizGrammar`, 2018. Viewed 6/10/22.

[89] A. Wu, Y. Wang, X. Shu, D. Moritz, W. Cui, H. Zhang, D. Zhang, and H. Qu. AI4VIS: Survey on Artificial Intelligence Approaches for Data Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2021.

[90] E. Wu, L. Battle, and S. R. Madden. The Case for Data Visualization Management Systems: Vision Paper. *VLDB*, 7(10):903–906, 2014.

[91] Y. Wu, J. M. Hellerstein, and A. Satyanarayan. B2: Bridging Code And Interactive Visualization in Computational Notebooks. In *Symposium on User Interface Software and Technology*, pp. 152–165. ACM, 2020. doi: 10.1145/3379337.3415851

[92] V. Zaytsev. Grammar Zoo: a Corpus of Experimental Grammarware. *Science of Computer Programming*, 98:28–51, 2015.

[93] J. Zong, D. Barnwal, R. Neogy, and A. Satyanarayan. Lyra 2: Designing Interactive Visualizations By Demonstration. *IEEE Transactions on Visualization and Computer Graphics*, 2021.