Design Spaces of Domain-Specific Languages

Comparing and Contrasting Approaches in PL and HCI

Jonathan Zong [*1*], **Josh Pollock** [†1*], **Dylan Wootton** [‡1] **and Arvind Satyanarayan** [§1]

[1] *Massachusetts Institute of Technology, Cambridge, MA, U.S.*
[*] *Equal contribution*

Abstract

A domain-specific language (DSL) design space describes a collection of related languages via a series of, often orthogonal, dimensions. While PL and HCI researchers have independently developed methods for working with design spaces, the communities have yet to fully benefit from each others' insights. In pursuit of new approaches informed by both PL and HCI, we first review existing approaches researchers employ to conceptualize, develop, and use design spaces in DSL design across the two disciplines. For example, HCI researchers, when developing interfaces backed by DSLs, often treat the design process as core to their research contributions and theory-building. In PL, researchers have explored formal approaches to design spaces that help automate design space exploration and provide powerful conceptual clarity to language design tradeoffs. We then discuss areas where the two fields share common methods and highlight opportunities for researchers to combine knowledge across PL and HCI.

Keywords: Domain-specific languages. Design space. Programming languages. Human-computer interaction.

1 Introduction

Designing a domain-specific language (DSL) requires a careful examination of a domain's problem space. From operator selection to the underlying language models, small design decisions can have a large impact on performance, expressivity, and learnability for a DSL. Each of these decisions produces a branching path in the overall process. Researchers, in pursuit of what they believe to be the optimal design for a set of goals, must navigate tradeoffs between alternatives using heuristics.

Researchers in both the Programming Languages (PL) and Human-Computer Interaction (HCI) communities leverage design spaces to describe DSLs, compare them against each other, and generate new designs. We define DSLs as languages that model a specific application domain. For example, Vega-Lite provides a language for representing data visualizations as mappings from data to visual properties of graphical marks [1]. To find commonalities between these fields, we use the term DSL to not only refer to textual specifications (i.e, things we typically think of as "programming languages") but also interfaces backed by compositional language primitives. Chasins et al. note that "PL-backed interfaces" often underlie direct manipulation interfaces that allow users to express programs by interacting with graphical elements [2]. For example, Lyra [3] is an interface that generates output comparable in expressiveness to the Vega-Lite DSL, from which Lyra's internal representations take inspiration.

Designing a good DSL is challenging, and requires both PL and HCI research expertise. Yet design space theory has proceeded largely independently in the two research communities. We review the methods and approaches each field uses to develop and evaluate design spaces, and find that despite a shared purpose in system building, the PL and HCI communities develop and discuss language design spaces differently. In HCI, researchers document the design process to articulate the rationale behind design decisions and tradeoffs. To refine a design space, some HCI researchers have proposed discussion-based methods including *critical reflection*. These methods connect insights from individual DSL designs to create broader theories, which reflect lessons learned in creating systems for a domain.

PLATEAU
12th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

*Email: jzong@mit.edu
†Email: jopo@mit.edu
‡Email: dwootton@mit.edu
§Email: arvindsatya@mit.edu

Such theories inform future design in the domain and are a step towards generalization across multiple domains. They then use design spaces to describe and compare existing languages and to generate new ideas for designs. In PL, formal methods have been created for reasoning and using design spaces. These often take the form of executable specifications described using formal semantics or code implementations. This formalism allows a design space to be rendered in code and then operationalized as a DSL of its own. Such design space DSLs give users the agency to pick multiple points in a DSL design space within the same program. When theory meets practice, PL implementers often discuss language design tradeoffs openly with their community in requests for comments (RFCs).

Drawing on these differences, we discuss opportunities for each community to leverage the others' techniques to aid in exploration and use of design spaces. PL and HCI tend to focus their efforts at different levels of detail or abstraction. These complementary focuses suggest possible research methods that incorporate insights from both formal abstractions and detailed analyses. We also identify opportunities to take discussion-based methods explored in both communities and integrate them more closely into the research process. Finally, we pose questions about approaches to DSL design spaces that future work in PL and HCI might address using knowledge from both fields.

2 Background: Design Spaces of Domain-Specific Languages in PL and HCI

Computing researchers who study software systems make progress on research questions by designing and evaluating systems such as DSLs and user interfaces [4]. This design process typically includes exploratory iterative prototyping as the researchers consider alternative designs — considering different operators to expose or interface interactions to afford.

Many evaluation methods for these prototypes are suited to answering questions about outcomes of an individual system–for example, a language's performance or an interface's usability. Broader questions about the design of systems, rather than a specific system itself, are difficult to answer by evaluation methods that focus on the behavior of an individual system. To think systematically about design decisions, researchers have turned to design spaces as a conceptual model for understanding a domain. Through design methods such as drafting and prototyping, researchers make progress toward design spaces by developing underlying knowledge of how best to design for the domain [5].

As a mental model, we find it useful to think about a design space analogously to a space of cooking recipes. The axes in a design space for bread recipes measure the quantity of component ingredients in a recipe (e.g. 2 cups of flour, a pinch of yeast, a dash of oil). Points in the space are different bread recipes with their location determined by the quantity of their constitutive ingredients. Such a space is useful in that it provides a framework for describing recipes, comparing them against others, and generating undiscovered options. From using the design space, the heuristics of baking (i.e. "three cups of flour for every cup of water") are the broader theory that can be developed when comparing different possible recipes against each other. More formally, we use the definition presented by Botero et al. [6]: a design space is a "space of potentials that the available circumstances afford for the emergence of new designs." Design spaces provide a systematic way to reason about the design decisions that computing researchers make.

While exploring a new domain's design space, both PL and HCI researchers may engage in DSL design. For PL, this often involves creating a textual DSL and evaluating its performance. For HCI, DSLs that support user interfaces can either be directly exposed as a software library [1] or exist in the business logic underlying an interface [7], [8]. In the latter case, the DSL is latent — every interaction in the system is an expression of a statement in the DSL, just in a graphical modality. More generally, we use the term DSL to refer to the language that a user interacts with — whether the language is represented using a graphical user interface or text.

Both HCI [1] and PL [9]–[11] communities have built DSLs and used design spaces to evaluate and contextualize their research contributions. In an example from early HCI literature on input devices, Card et al. describe the design space of input devices with two dimensions [12]. The first is the physical property sensed by the input device, such as absolute or relative position or force. The second is the linear or rotary spatial axis that the input device senses movement along. These dimensions parameterize the space enabling input device designs to be represented as points in the

space. For instance, a computer mouse senses relative position along the linear x and y axes whereas a scroll ball works through rotational axes. Card et al. uses this space to characterize other input devices including joysticks, pads, and thumbwheels. In the development of Glinda, a DSL-backed interface [7], the authors differentiate Glinda from prior work in terms of how the system maintains control on notebook cell execution. Such comparisons refer to design space axes help to contextualize how new interfaces fall into the existing body of literature. More recently, PL researchers have created systematic frameworks for comparing languages by their mixed-type semantics [13] and comparing build systems along many dimensions in a design space [14]. Such contributions represent a similar structure to that of HCI contributions; these design spaces are used to describe, compare, and generate new types of designs.

3 How PL and HCI Approach Design Spaces

3.1 HCI Approaches to Design Spaces

Because the field of HCI is centrally concerned with how people interact with computational artifacts, HCI researchers are trained to consider the needs and goals of the users of their systems. This disciplinary commitment shapes how HCI researchers understand design spaces. For HCI, design spaces are not only useful for modeling the features systems implement; they are also frameworks for thinking about how a system's design decisions encode assumptions about who the user is, what their goals are, and how design decisions facilitate those goals. In this section, we explore three key approaches that HCI researchers use to build knowledge about DSL design spaces: documenting design rationale to gather initial evidence about design decisions, applying critical reflection methods to create design spaces from point systems, and developing theory for evaluating design spaces.

3.1.1 Design Exploration as a Research Contribution

When HCI researchers design languages and interfaces, they often begin by articulating what user goals researchers aim to support and what subjective values they want their designs to reflect. Because DSL designs reflect the outcomes of a designer's decision-making process, the field of HCI has sought to understand "the goals and politics of human-designed devices" [15]. Writing about the assumptions behind design decisions — for instance, about who the user is and what their goals are — provides contextual information that informs how researchers evaluate contributions [16].

The design and evaluation process involves iteratively exploring a range of designs. During this process, researchers reason about tradeoffs, refine their understanding of how designs achieve end goals for users, and think about how designs encode assumptions and values that might have social consequences. But while the final design reflects the outcome of these decisions, it "does not [preserve] the thinking and reasoning which went into its creation" [16]. For this reason, HCI researchers usually discuss the design process and share early prototypes as part of research papers [17], [18].

HCI researchers use the term *design rationale* to refer to knowledge about the motivations, reasoning, and context behind decisions made during the design process [19]. By documenting design rationale in research papers, authors create useful evidence that can inform future designs in the same domains. Even if a researcher discards a particular language design, other researchers could learn from the reasoning behind that decision. For example, a design that was unsuited to a certain set of user goals and constraints could be useful in a different context. Carroll and Rosson write that design rationale takes "the terms and relations of the application domain, and [grounds] them in design tradeoffs and decisions" [19]. Starting from design rationale, HCI researchers can develop vocabulary to compare language designs in the same domain in terms of specific decisions. These conversations in the research community can later inform the creation of design space dimensions.

3.1.2 Critical Reflection: A Method for Creating Design Spaces

Researchers make progress on understanding the space of DSLs in a domain by designing languages and implicitly building evidence for theories by documenting design rationale. But once many DSL designs exist in a domain, researchers have opportunities to more explicitly compare and contrast fully developed languages. To formalize analysis of existing DSLs in HCI, researchers have proposed *critical*

reflection as a method for qualitative analysis of designs in a discussion-based format [20].

Critical reflection (CR) involves the designers of multiple DSLs in the same domain. The designers collectively reflect on the shared objectives of their systems and discuss how each system embodies them. During CR, designers articulate questions they asked during the design process (e.g. "how do we deal with object inheritance"). Through discussions of the decisions' resulting impact, designers improve their understanding of how trade-offs affect progress towards shared objectives (e.g. expressivity or performance). For example, in "Critical Reflections on Visualization Authoring Systems," Satyanarayan et al. reflect on the design process of three visualization authoring interfaces backed by DSLs [20]. In their paper, the authors discuss important decision points in the design of all three systems, surface shared assumptions implicit in these systems, and further structure the design space to identify promising future work.

Though the use of critical reflections has produced promising results for understanding the design of visualization systems, there are not yet many examples of CR contributions in the HCI literature. In contrast to other forms of critique and user feedback, which can occur throughout the design process, CR occurs retrospectively after multiple designs have been created. It is also a time-intensive process requiring effort and commitment from multiple experts. These characteristics are strengths of CR that enable researchers to more precisely connect design decisions to design outcomes. However, they also make it difficult for CR to be put into commonplace usage. Open questions remain about how to do critical reflection in domains where researchers have not produced multiple fully-developed DSLs, and how to properly credit and compensate expert participants. But because CR is inspired by the use of critique in other design-related fields, including art and architecture, promising opportunities exist in HCI to learn from how other fields do critique [21].

3.1.3 Theories for Evaluating Design Spaces

When researchers create design spaces, they define design dimensions with which to describe the space. Different design spaces could model designs in the same domain using different dimensions. How can researchers evaluate the usefulness of a particular way of defining a design space?

Beaudouin-Lafon writes that models and frameworks for designers of interactive systems can be understood along three criteria: descriptive power, evaluative power, and generative power [22]. *Descriptive power* refers to a framework's ability to provide useful language for describing the features of individual systems. For design spaces, this means being able to successfully map a broad range of designs onto points in the space. If the design space dimensions don't correspond well to important distinguishing features of the design, this could be a sign that the design space does not have sufficient descriptive power to be useful for analysis. *Evaluative power* refers to a framework's utility in comparing across different designs. Situating systems as points in a design space allows researchers to use design dimensions to articulate differences between systems along those dimensions. Sometimes, dimensions have an evaluative direction, where systems on one end of the spectrum are clearly preferable to systems on the other end. Other times, design dimensions represent knowledge about decisions in the design process where two systems could reasonably differ based on their goals and intended users. In both cases, decomposing the evaluation of designs into distinct dimensions clarifies the purpose of evaluation. *Generative power* refers to how well a framework suggests the existence of possible designs that haven't been explored yet. Plotting many designs as points in a design space can reveal gaps where there are no points. Thinking about the design features that correspond to missing points can suggest new directions for exploration. It can also yield useful knowledge about why those designs may be difficult or not viable to make.

The three powers have been used in HCI both to analyze point systems and to analyze conceptual frameworks that model design spaces of systems. In a keynote on toolkits for machine learning interpretability, Satyanarayan uses the three powers to analyze how different interpretability systems allow users to inspect model predictions to generate insights, make comparisons, and identify patterns [23]. Because Beaudouin-Lafon's three powers are useful for analysis at different levels of abstraction — from point systems to design spaces — they serve as useful conceptual tools for building point observations into theories that advance researchers' understanding over time.

3.2 PL Approaches to Design Spaces

Programming language researchers are often concerned with formal properties of the systems they build. Design spaces in PL research thus often come in the form of more *formalized* design spaces. Researchers operationalize these design spaces by constructing formal models, sometimes in code. They use these models to prove properties about points in the space, like whether a DSL prevents memory leaks, or even to write executable prototypes in a general purpose language. Such prototypes help validate the design space since they can be tested using existing language tools. Since users can interact with them, these prototypes also allow researchers to experiment directly with points in the design space. In fact, researchers have explored realizing a design space *itself*, not just individual points, as an executable prototype. This grants users the flexibility to switch between different points in the design space. In the realm of PL implementation, simplified formal models meet the realities of integration with real systems. Design spaces in language implementation are often explored in requests for comments (RFCs), where concerned parties discuss how best to implement language features so they integrate well with existing language quirks and constructs.

3.2.1 Formal Models of Design Spaces

PL researchers often strive for design spaces that capture the simple "essence" of a set of design tradeoffs. These spaces often come in the form of small, executable specifications described using formal semantics or code implementations. One notable recent example is "Build Systems á la Carte," which provides "a systematic, and executable, framework for developing and comparing build systems, viewing them as related points in landscape rather than as isolated phenomena" [14].

In his talk on the paper, Peyton Jones argues for the importance of studying simplified models of systems when defining a design space:

> You can't build a real build system in 20 lines of code. Real build systems [...] do tons of stuff that we're not modeling at all. There's parallelism; there's non-determinism; there's dealing with failure and recovery; and there's trying to distribute all of this across your data center.
>
> So it's only a model, but that's a strength as well as a weakness. Because it's small it's intellectually tractable... [Y]ou can invent new things that you hadn't been able to do at scale before, and hopefully... you can scale it up. [24]

Reducing a collection of systems to their essences can reveal connections that implementation details hide. For example, Mokhov et al.'s build system [14] design space allows researchers to compare Excel and Make — two different pieces of software with very different user experiences. A build system must execute a series of tasks with some dependencies between them. Both Make and Excel use the "dirty bit" strategy to track which tasks must be rebuilt, but Make schedules tasks topologically while Excel uses a restarting algorithm that aborts a task if one of its dependencies is out of date. Restarting is less efficient, since it may build the same task multiple times, but it allows dependencies to be dynamic. By making large leaps smaller, abstract design spaces allow researchers to explore radically different design alternatives.

Greenman applies a similar approach in his dissertation, *Deep and Shallow Types* [13]. Greenman constructs a design space of mixed-type languages. These languages, including TypeScript and mypy, support both typed and untyped code in the same file or codebase. As with Mokhov et al., Greenman elides many implementation details. Crucially Greenman constructs models of many mixed-type systems as extensions to the same language semantics.

These simplified models allow Greenman to define formal properties that separate clusters of points in the space by how untyped and type code interact with each other. One property Greenman defines is called *complete monitoring*, which is a mixed-type system that strongly enforces a type signature, like `Int -> Int`, even if the input is untyped. TypeScript, for example, does not satisfy the complete monitoring property, since untyped data is not checked at runtime.

Formal properties like complete monitoring are much easier to discover, state, and prove in a simplified environment than they are for real languages, which usually have more primitive constructs

and edge cases. Nevertheless, Greenman uses these formal properties to analyze the idealized versions of real-world mixed-type systems.

Eliding complexity in favor of intellectual tractability enables many techniques in PL. Abstracting away implementation details lets researchers wrestle with formal properties that can provide precise structure to a design space.

3.2.2 Formal Design Spaces Can Produce Flexible Languages

Another advantage of formal design space descriptions is that they can be rendered in code. For example, Mokhov et al. wrote their build system design space in Haskell. Unlike hand-written formal specifications, a general-purpose programming languages provides an unambiguous, machine-readable description of a design space. In addition to checking a researcher's work by ensuring a definitions typecheck, languages like Haskell readily expose abstractions. Mokhov et al. were able to reuse existing typeclasses like `Applicative` and `Monad` to describe different kinds of build system tasks.

But there is an even more exciting benefit to writing a design space in code. *The design space itself can operationalized.* That is, a formal design space can be turned into a DSL that allows the *user* to pick a point in the space as they see fit. Often, rather than limiting the user to a single point throughout their entire program, a design space DSL allows a user to select points *locally* so they can take advantage of multiple designs.

For example, there is a four-decade-old debate [25]–[30] in compiler design about whether one program representation, continuation-passing style (CPS), is better than another, direct style, for writing program optimizations. In 2019, Cong et al. published a paper arguing that designers don't need to choose [31]. You can have "as much or as little CPS as you want, when you want it." To achieve flexibility, the authors first explore the existing design space of program representations, and then combine Kennedy's CPS approach [29] with Maurer et al.'s direct style approach [30] to achieve a *new* program representation capable of both. By reifying the design space in a single DSL, Cong et al. allow compiler writers to use mix and match styles in the same program.

Greenman is able to use his design space analysis to achieve a similar result for mixed-type systems. In addition to just partitioning the design space using formal properties, Greenman leverages his gradual typing formalism to construct a way to *mix* different typing disciplines together in the same codebase. Specifically, Greenman shows how to systematically integrate deep, shallow, and untyped code in the same codebase. Reasoning about the interfaces between the three requires a deep understanding of the characteristic properties of each, which Greenman provided with his formal analysis of the design space. By constructing a language that allows for multiple mixed-typing disciplines in the same codebase, users gain more freedom to choose between the expressiveness of untyped code and the safety of typed code. In fact, because his language allows for multiple typing disciplines, Greenman is able to propose a new workflow for migrating untyped code to typed code: "Use shallow types when converting an untyped application and switch to deep types after the boundaries stabilize." This workflow is only possible when a design space is reified in a single DSL.

3.2.3 Public Discussions of Language Design in Requests for Comments

While the previous two sections have focused on the advantages of simple formal models, when it comes time to implement these abstract ideas in real-world languages complexity quickly resurfaces. Implementing a new feature in a programming language requires careful consideration of how that feature will interact with existing ones, how easy it is to learn, and even how it could affect parsing.

In standards and commercial projects, language designers have an established practice of creating *requests for comments (RFCs)* to explore the design space for new features. RFCs are structured proposals for programming language extensions and revisions. They explore the existing design space by referring to existing theory and other language implementations while exploring design alternatives. RFCs represent repositories that serve communities interested in learning how to extend a system or further develop in a given area [32].

RFCs are effective for capturing information about design decisions and tradeoffs. They also provide archives for how and why decisions were made. RFCs help develop nascent theory about PL

design decisions in the wild, but they can make it difficult to step back and analyze larger trends and insights about effective approaches to language features in general.

4 Bridging PL and HCI

By reviewing different approaches to conceptualizing DSL design in PL and HCI, we begin to identify important commonalities. Both fields use empirical observations gleaned from fully designed point systems to build conceptual models of design spaces. Both use discussion-based methods to generate insight about design decisions, though these methods take place in different contexts.

The two fields also differ in important ways. HCI has well-developed ideas about using design rationale to build conceptual models about design, and useful theories for evaluating those conceptual models. PL researchers could benefit from adopting techniques and theories such as critical reflection and Beaudouin-Lafon's three powers to structure the informal discussion and reflection that already happens in PL communities. On the other hand, PL has a propensity for developing simplified models of systems with precise formal properties. HCI would benefit from thinking more clearly about language design, rather than entangling language design with interface implementation details. As Chasins et al. argue, "PL and HCI researchers can integrate their complementary expertise to advance goals that matter in both communities" [2]. In the following discussion, we think through ways to take advantage of this complementary expertise.

4.1 Combine Detailed and Abstract Approaches

In this paper we have analyzed different approaches to design space construction. Broadly, HCI researchers favor detailed system models that retain complexity that is relevant to understanding user experience. Writing about their experience of CR over visualization authoring systems, Satyanarayan et al. noted that because there can be "significant inconsistencies between the abstractions [systems expose], all stakeholders needed to actively participate in extracting and mapping intricate *low-level details*" [20]. On the other hand, PL researchers are partial to more abstract models that are amenable to formal reasoning and proof. As we observed in Mokhov et al.'s build systems work and Greenman's work on mixed types, language designers use abstract models to state and prove properties about clusters of points in a DSL design space. By eliding detail, abstract spaces shrink the distance between systems, which allows for comparisons across languages that might be initially seem unrelated. Similarly, such spaces afford new designs that may be radically different as well.

Since DSLs bridge PL theory and the design of human interfaces, how can DSL researchers get the best of both worlds? We offer a few suggestions for how these approaches might be combined.

The strengths of a detailed design space come from its closeness to real implementations. Meanwhile the strength of an abstract one comes from its ability to generalize over seemingly-different implementations that share underlying primitives. This difference suggests an approach to design space exploration that moves between detailed and abstract analysis could build clearer understanding.

We are inspired by Coblenz et al.'s [33] process for designing programming languages with techniques that iterate between HCI and PL approaches. In the PLIERS process, researchers first use HCI methods to identify user needs, then iterate between identifying key theoretical concepts underlying a potential design and evaluating the usability of those concepts in the context of a larger system.

Design space approaches could be integrated into the PLIERS process. Instead of starting with a need-finding study, a researcher might begin by constructing a detailed design space of many related DSLs. For example, in the study of build systems one might reflect on the design rationale of the underlying algorithms, tradeoffs between spreadsheet and text file interfaces, and how dependencies are specified. Next, the researcher could isolate individual features and abstract them so they become amenable to PL-style formal reasoning. This process is similar to the core calculus development in PLIERS; however, the result might be a formal design space rather than the core of a DSL. Finally, once the individual components of a system have been analyzed abstractly, then can be re-contextualized in the original systems with all their complexity. Now aided by the power of precise formal properties from their abstractions, researchers can understand detailed design spaces at a deeper level.

4.2 Synthesize Knowledge from Both RFCs and Critical Reflection

Because RFCs and critical reflection both rely on discussion to generate new insights, we argue there are opportunities for productive methodology exchange between these two approaches to documenting and reflecting upon design decisions.

PL researchers use RFCs to explore alternative design decisions for a language feature. Though language experts most commonly engage with RFCs, they offer opportunities for anyone (including language users) to recommend suggestions. As RFCs document development as it happens, they capture design decisions as they occur rather than in retrospect. The primary output of the RFCs are language features, but RFCs also serve as a learning resource for individuals interested in the domain. While this forum for evaluating design decisions generates important insight about language design, these RFCs take place outside of formal academic processes like peer review. As a result, PL communities don't treat the intellectual work that happens in RFCs as research contributions.

In HCI, discussions on the design rationale of a system are a part of a typical research contribution. Such a focus has allowed for the creation of evaluation techniques such as CR. CR has been developed as a method for informing the design space and synthesizing knowledge from individual point designs in a design space. Such discussions are typically closed, limited to authors of similar systems who are able to discuss the design alternatives they considered. These discussions help to illuminate the design space axes and can inform broader theories about how to navigate tradeoffs in it.

We believe both communities can benefit from these methods. For instance, although HCI papers report design rationale and discuss a select number of design decisions, using a documented process like an RFC could more accurately capture the evolution of a researchers' thought process. Though a typical RFC process can be labor-intensive and require a user to dig for insights embedded in larger discussions, we suspect a modified form of RFCs could provide an opportunity to document a DSL design process outside of the constraints of a conference paper and provide a trove of information for interface developers to learn from. Conversely, while these design discussions are rarely viewed as a part of the contribution in PL, they have contributed to the development of broader theories in HCI, and encouraging researchers to include them in publications could aid in generalizing lessons learned from developing in a given domain.

In PL, we've seen methods similar to CR used but were not explicitly described as such. For example, Neil Mitchell, one of the authors of "Build Systems à la Carte," wrote a blog post discussing the research process behind the paper [34]. In this post, Mitchell describes the authors' experiences receiving feedback from experts in the community after sharing their draft online . They attracted comments on their draft that responded to their framework in depth, including from the creators of systems who were able to improve the framework's description of their system. By engaging with the draft and applying their own expertise to draw out new insights about the proposed framework, the online community was participating in an informal process similar to critical reflection. The online community identified the goal of the work, and arrived at a detailed enough descriptive understanding of the work to understand how it was successful or unsuccessful. In turn, the authors of the work came to a better understanding of their own specific framework, and a better general understanding of what details about build systems are interesting to model. The PL and HCI research communities could benefit from turning these informal discussions into more structured methods like CR to more closely integrate them into the research process.

Critical reflections on DSLs are a specific form of expert analysis, which several researchers have used to evaluate DSLs [33], [35]. We claim that by gathering feedback, not just from DSL experts, but from experts *who have built similar systems themselves*, researchers may glean a deeper understanding of their own work and how it fits into a larger space of related systems.

4.3 Open Questions for Design Spaces

In reviewing the use of design spaces for language design in PL and HCI, we encountered some questions for future work. While our discussion focuses on design spaces that represent possible language designs, researchers have used design spaces to understand systems at various levels of abstraction. For example, grammars of graphics such as Vega-Lite are DSLs that formally represent

a design space of visualizations [1]. Indeed, most DSLs can be thought of as design spaces over the expressible programs in that language. We note that as DSLs move up levels of abstraction—from programs, to languages, to classes of language–there are tradeoffs between generality and Beaudouin-Lafon's three powers.

For instance, many DSL design spaces from the PL literature we have examined so far lend themselves to formal modeling with high generative power. By nature of their formality, the design dimensions tend to be specific and highly separable, which allows them to be manipulated independently. Mokhov et al. are able to generate build systems just by mixing and matching scheduling and rebuilding techniques [14]. But more general design spaces seem to necessarily lack these properties. For example, Green's cognitive dimensions of notation is a conceptual framework featuring orthogonal design dimensions meant to apply to "many types of language—interactive or programming, high or low level, procedural or declarative, special purpose or general purpose" [36]. Because its dimensions are so generally applicable, they are also impossible to represent as a formal model that can mechanistically output new languages. Future work on design spaces might uncover underlying structure or hierarchy in the design space of design spaces as researchers encounter tradeoffs as they develop new conceptual models to guide design.

Acknowledgements

We would like to thank the MIT Visualization Group; PLATEAU workshop organizers, participants, and mentors; and our anonymous reviewer. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1745302 and by NSF Grant III-1900991.

References

[1] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-Lite: A Grammar of Interactive Graphics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 341–350, Jan. 2017, ISSN: 1941-0506. DOI: 10.1109/TVCG.2016.2599030.

[2] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine, "PL and HCI: Better Together," en, *Communications of the ACM*, vol. 64, no. 8, pp. 98–106, Aug. 2021, ISSN: 0001-0782. [Online]. Available: https://cacm.acm.org/magazines/2021/8/254314-pl-and-hci/fulltext (visited on 09/19/2021).

[3] J. Zong, D. Barnwal, R. Neogy, and A. Satyanarayan, "Lyra 2: Designing Interactive Visualizations by Demonstration," en, *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 304–314, Feb. 2021. DOI: 10.1109/TVCG.2020.3030367.

[4] Y.-K. Lim, E. Stolterman, and J. Tenenberg, "The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas," en, *ACM Transactions on Computer-Human Interaction*, vol. 15, no. 2, pp. 1–27, Jul. 2008, ISSN: 1073-0516, 1557-7325. DOI: 10.1145/1375761.1375762. [Online]. Available: https://dl.acm.org/doi/10.1145/1375761.1375762 (visited on 09/16/2021).

[5] K. Halskov and C. Lundqvist, "Filtering and Informing the Design Space: Towards Design-Space Thinking," en, *ACM Transactions on Computer-Human Interaction*, vol. 28, no. 1, pp. 1–28, Feb. 2021, ISSN: 1073-0516, 1557-7325. DOI: 10.1145/3434462. [Online]. Available: https://dl.acm.org/doi/10.1145/3434462 (visited on 09/16/2021).

[6] A. Botero, K.-H. Kommonen, and S. Marttila, "Expanding Design Space: Design-In-Use Activities and Strategies," en, *Design Research Society International Conference*, p. 13, Jan. 2010.

[7] R. A. DeLine, "Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language," en, in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, Yokohama Japan: ACM, May 2021, pp. 1–11, ISBN: 978-1-4503-8096-6. DOI: 10.1145/3411764.3445267. [Online]. Available: https://dl.acm.org/doi/10.1145/3411764.3445267 (visited on 09/16/2021).

[8] J. Echeveste, A. Cont, J.-L. Giavitto, and F. Jacquemard, "Operational semantics of a domain specific language for real time musician–computer interaction," en, *Discrete Event Dynamic Systems*, vol. 23, no. 4, pp. 343–383, Dec. 2013, ISSN: 0924-6703, 1573-7594. DOI: 10.1007/s10626-013-0166-2. [Online]. Available: http://link.springer.com/10.1007/s10626-013-0166-2 (visited on 09/16/2021).

[9] M. Coblenz, J. Aldrich, B. A. Myers, and J. Sunshine, "Interdisciplinary Programming Language Design," en, *New York*, p. 15, 2018.

[10] U. Zdun and M. Strembeck, "Reusable Architectural Decisions for DSL Design," en, *CEUR Workshop Proceedings*, vol. 566, p. 37, 2019.

[11] A. Stefik and S. Siebert, "An Empirical Investigation into Programming Language Syntax," en, *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 1–40, Nov. 2013, ISSN: 1946-6226. DOI: 10.1145/2534973. [Online]. Available: https://dl.acm.org/doi/10.1145/2534973 (visited on 11/22/2021).

[12] S. K. Card, J. D. Mackinlay, and G. G. Robertson, "A morphological analysis of the design space of input devices," en, *ACM Transactions on Information Systems*, vol. 9, no. 2, pp. 99–122, Apr. 1991, ISSN: 1046-8188, 1558-2868. DOI: 10.1145/123078.128726. [Online]. Available: https://dl.acm.org/doi/10.1145/123078.128726 (visited on 09/16/2021).

[13] B. Greenman, "Deep and shallow types," en, Ph.D. dissertation, Northeastern University, 2020. DOI: 10.17760/D20398329. [Online]. Available: http://hdl.handle.net/2047/D20398329.

[14] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems à la carte," en, *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 1–29, Jul. 2018, ISSN: 2475-1421. DOI: 10.1145/3236774. [Online]. Available: https://dl.acm.org/doi/10.1145/3236774 (visited on 09/08/2021).

[15] K. Shilton, "Values and Ethics in Human-Computer Interaction," English, *Foundations and Trends® in Human–Computer Interaction*, vol. 12, no. 2, pp. 107–171, Jul. 2018, Publisher: Now Publishers, Inc., ISSN: 1551-3955, 1551-3963. DOI: 10.1561/1100000073. [Online]. Available: https://www.nowpublishers.com/article/Details/HCI-073 (visited on 02/07/2021).

[16] A. MacLean, V. Bellotti, and S. Shum, "Developing the Design Space with Design Space Analysis," en, in *Computers, Communication and Usability: Design issues, research and methods for integrated services*, ser. North Holland Series in Telecommunication, P.F. Byerley, P.J. Barnard, and J. May, Eds., Amsterdam: Elsevier, 1993, pp. 197–219.

[17] A. K. Hopkins, M. Correll, and A. Satyanarayan, "VisuaLint: Sketchy In Situ Annotations of Chart Construction Errors," en, *Computer Graphics Forum*, vol. 39, no. 3, pp. 219–228, Jun. 2020, ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.13975. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/cgf.13975 (visited on 09/19/2021).

[18] C. Nobre, D. Wootton, L. Harrison, and A. Lex, "Evaluating Multivariate Network Visualization Techniques Using a Validated Design and Crowdsourcing Approach," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–12, ISBN: 978-1-4503-6708-0. [Online]. Available: https://doi.org/10.1145/3313831.3376381 (visited on 09/19/2021).

[19] J. M. Carroll and M. B. Rosson, "Design Rationale as Theory," in *HCI Models, Theories, and Frameworks*, Elsevier Inc., Apr. 2003, pp. 431–461, ISBN: 978-1-55860-808-5. DOI: 10.1016/B978-155860808-5/50015-0. [Online]. Available: http://www.scopus.com/inward/record.url?scp=84902232794&partnerID=8YFLogxK (visited on 09/14/2021).

[20] 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*, vol. 26, no. 1, pp. 461–471, Jan. 2020, Conference Name: IEEE Transactions on Visualization and Computer Graphics, ISSN: 1941-0506. DOI: 10.1109/TVCG.2019.2934281.

[21] J. Bardzell, "Interaction criticism: An introduction to the practice," *Interacting with Computers*, vol. 23, no. 6, pp. 604–621, Nov. 2011, ISSN: 0953-5438. DOI: 10.1016/j.intcom.2011.07.001. [Online]. Available: https://doi.org/10.1016/j.intcom.2011.07.001 (visited on 09/14/2021).

[22] M. Beaudouin-Lafon, "Designing interaction, not interfaces," en, in *Proceedings of the working conference on Advanced visual interfaces - AVI '04*, Gallipoli, Italy: ACM Press, 2004, p. 15, ISBN: 978-1-58113-867-2. DOI: 10.1145/989863.989865. [Online]. Available: http://portal.acm.org/citation.cfm?doid=989863.989865 (visited on 09/19/2021).

[23] Arvind Satyanarayan, *From Tools to Toolkits: Towards more Reusable, Composable, and Reliable Machine Learning Interpretability*, en, 2021. [Online]. Available: https://vimeo.com/590958970.

[24] ICFP Video, *Build Systems à la Carte*, Oct. 2018. [Online]. Available: https://www.youtube.com/watch?v=BQVT6wiwCxM (visited on 09/19/2021).

[25] G. L. Steele Jr, "Rabbit: A compiler for Scheme," 1978.

[26] N. Adams, D. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin, "Orbit: An optimizing compiler for Scheme," *ACM SIGPLAN Notices*, vol. 21, no. 7, 1986.

[27] A. W. Appel, "Compiling with Continuations," *Cambridge University Press*, vol. 2, pp. 46–55, 1992.

[28] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993, pp. 237–247.

[29] A. Kennedy, "Compiling with continuations, continued," in *Proceedings of the 12th ACM SIGPLAN*, 2007, pp. 177–190.

[30] L. Maurer, P. Downen, Z. M. Ariola, and S. Peyton Jones, "Compiling without continuations," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 482–494.

[31] Y. Cong, L. Osvald, G. M. Essertel, and T. Rompf, "Compiling with continuations, or without? whatever.," en, *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–28, Jul. 2019. DOI: 10.1145/3341643. [Online]. Available: https://dl.acm.org/doi/10.1145/3341643.

[32] S. D. Crocker, "Learning to network," *IEEE Annals of the History of Computing*, vol. 41, no. 2, pp. 42–47, Apr. 2019, ISSN: 1934-1547. DOI: 10.1109/MAHC.2019.2909848.

[33] M. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers, "PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design," en, *arXiv:1912.04719 [cs]*, Aug. 2020, arXiv: 1912.04719. [Online]. Available: http://arxiv.org/abs/1912.04719 (visited on 09/01/2021).

[34] N. Mitchell, *Inside the paper: Build Systems a la Carte*, Jul. 2018. [Online]. Available: http://neilmitchell.blogspot.com/2018/07/inside-paper-build-systems-la-carte.html (visited on 09/08/2021).

[35] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," en, *Computer*, vol. 49, no. 7, pp. 44–52, Jul. 2016, ISSN: 0018-9162, 1558-0814. DOI: 10.1109/MC.2016.200. [Online]. Available: https://ieeexplore.ieee.org/document/7503516/ (visited on 11/24/2021).

[36] Thomas RG Green, "Cognitive dimensions of notations," *People and computers V*, 1989. [Online]. Available: https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/papers/Green1989.pdf.