How Domain Experts Use an Embedded DSL

LISA RENNELS, University of California at Berkeley, USA
SARAH E. CHASINS, University of California at Berkeley, USA

Programming tools are increasingly integral to research and analysis in myriad domains, including specialized areas with no formal relation to computer science. Embedded domain-specific languages (eDSLs) have the potential to serve these programmers while placing relatively light implementation burdens on language designers. However, barriers to eDSL use reduce their practical value and adoption. In this paper, we aim to deepen our understanding of how programmers use eDSLs and identify user needs to inform future eDSL designs. We performed a contextual inquiry (9 participants) with domain experts using Mimi, an eDSL for climate change economics modeling. A thematic analysis identified five key themes, including: the interaction between the eDSL and the host language has significant and sometimes unexpected impacts on eDSL user experience, and users preferentially engage with domain-specific communities and code templates rather than host language resources. The needs uncovered in our study offer design considerations for future eDSLs and suggest directions for future DSL usability research.

CCS Concepts: • **Human-centered computing** → **Empirical studies in HCI**; • **Software and its engineering** → **Designing software**.

Additional Key Words and Phrases: embedded domain-specific languages, user experience, contextual inquiry, need finding, usability

ACM Reference Format:
Lisa Rennels and Sarah E. Chasins. 2023. How Domain Experts Use an Embedded DSL. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 275 (October 2023), 32 pages. https://doi.org/10.1145/3622851

1 INTRODUCTION

Experts across a diverse range of fields have come to rely on programming and computation. We see this trend playing out across myriad domains, including earth sciences and climate change [Balaji et al. 2018; Easterbrook 2010; Easterbrook and Johns 2009; Williams 2014]. For domains like these, in which practitioners typically lack formal computer science education, the usability and learnability of available languages can be substantial pain points. These users typically do not share the body of computing-related background knowledge we expect from professional or formally trained software engineers. However, domain experts within a given field often *do* share common problem formulations, vocabulary, and knowledge. Domain-specific languages (DSLs) are a long-standing and popular strategy for meeting the needs of such domain experts.

Embedded domain-specific languages (eDSLs), including libraries, are a particularly popular approach. Embedded DSLS *embed* new constructs within a general-purpose language (GPL), adding a thin layer of domain-specific elements specialized for a particular community's needs. In contrast to developing an entirely new, standalone language, embedding is efficient for the designer, although the host language may constrain and complicate the DSL [Cavé et al. 2010; Freeman and Pryce 2006; Hudak 1998; Mernik et al. 2005]. (Section 7.1 of our Related Works offers a more detailed discussion

Authors' addresses: Lisa Rennels, lrennels@berkeley.edu, University of California at Berkeley, Berkeley, USA; Sarah E. Chasins, schasins@cs.berkeley.edu, University of California at Berkeley, Berkeley, USA.

2475-1421/2023/10-ART275
https://doi.org/10.1145/3622851

of how eDSLs' lower implementation burdens have made the embedded approach popular with DSL designers). From the perspective of the DSL *user*, an eDSL may provide a richer and more extensible tool than a standalone DSL.

While eDSLs can be immensely valuable, user experience challenges may make eDSLs less appealing to *users* than to *designers* [Gray et al. 2008]. One empirical study of 10 DSL implementations (nine non-embedded DSLs and one eDSL) found that while the eDSL ranked best in *implementation effort*, it ranked second worst in *end-user effort* [Kosar et al. 2008]. This suggests an opportunity for language designers to improve user experience via a better understanding of eDSL user needs.

User experience analyses of eDSLs naturally differ from analyses of GPLs and even non-embedded DSLs. For example, the effect on users of the relationship between an eDSL and its host language are unique to the embedded approach and demand fresh analyses. Similarly, user experience analyses of domain experts using programming languages differ from analyses of practiced programmers. For example, priorities and metrics for successful language use may differ across these different settings [Barišić et al. 2011].

This paper describes a study of eDSL user experience in which all participants are experts in the eDSL's domain. Our study advances our knowledge both of the overall eDSL user experience and more specifically domain experts' user experience with eDSLs.

To date we have only a small body of literature observing how users *actually* interact with embedded DSLs, and an even smaller subset observing how domain experts interact. (See Section 7.2.2 for a summary of this work to date.) By contributing to this line of research, we hope to support the community in improving eDSL usability and usefulness, making eDSLs attractive not only for DSL designers but also for a broad array of users.

We conducted a contextual inquiry, employing observations and semi-structured interviews to:

(1) directly observe domain experts' user experience with a climate change economics eDSL
(2) examine the impact of eDSL design and implementation decisions on user experience
(3) suggest design considerations for future eDSLs

Our findings fall into five core themes: (i) impacts of the host language, (ii) impacts of the interaction between host and embedded language, (iii) eDSL-specific communities, (iv) eDSL-specific resources, and (v) integration of multiple tools via the host language. Within each high-level theme, we observed a range of behaviors and patterns. For example, although many eDSL design guidelines suggest blurring the line between an eDSL and its host language, we observed that blurring this line made debugging tasks difficult for participants, especially for novice programmers. Another example: participants preferred to seek eDSL-specific resources, including eDSL-specific communities and eDSL-specific tutorials, even for needs that could have been satisfied by StackOverflow or language-specific resources. Based on our findings, we suggest (i) a set of design considerations for future eDSL designers and (ii) directions for future eDSL user experience research.

2 BACKGROUND

Here we briefly introduce key terms and the particular eDSL we used in our study.

2.1 Key Terms

Developers vs. Users. Throughout this paper, we will use the term *developer* or *designer* to refer to individuals designing and implementing languages, DSLs, and eDSLs. We will use the term *user* or *end user* to refer to individuals writing programs with those languages, DSLs, and eDSLs.

User Experience, Usability, and Learnability. We will use the umbrella term *user experience* to discuss many aspects of programmers' interactions with languages, programming tools, language communities, and programming environments. We will use the term *usability* specifically for

discussions of whether users can easily accomplish particular goals with a given tool. We will use the term *learnability* for discussions of whether tools are easy or difficult for new users to learn.

2.2 Mimi, an eDSL for Integrated Assessment Models

This paper presents a contextual inquiry of how domain experts use Mimi, an eDSL for Integrated Assessment Models (IAMs). IAMs are used to calculate the social cost of carbon dioxide (SC-CO_2), an economic metric employed extensively to inform a range of climate policy decisions. The SC-CO_2 represents the net economic effects of emitting one additional metric tonne of carbon dioxide into the atmosphere. Applications of the SC-CO_2 include calculating the value of taxes for carbon tax regulation, utilities resource planning, and cost-benefit analysis made on municipal and federal levels. The 2018 Nobel Memorial Prize in Economic Sciences honored Professor William Nordhaus, who pioneered these macroeconomic models in the early 1980s [Nordhaus 1982].

Until recently, IAMs were implemented via a fragmented conglomeration of many different programming languages, APIs, hosting platforms, and other tools. Interacting with these miscellaneous programming tools makes IAM analyses daunting for even experienced domain experts. Analyzing one, let alone comparing several models requires significant learning and time-consuming harmonization work. Researchers and other interested parties must write their own code to carry out crucial steps like uncertainty analysis and sensitivity analysis, with the result that practitioners often skip it or perform it haphazardly. In 2017 a seminal report by the National Academies of Sciences, Engineering, and Medicine recommended the development of "an open-source, computationally efficient, publicly accessible, and clearly documented computational platform" to help remedy these problems [NASEM 2017].

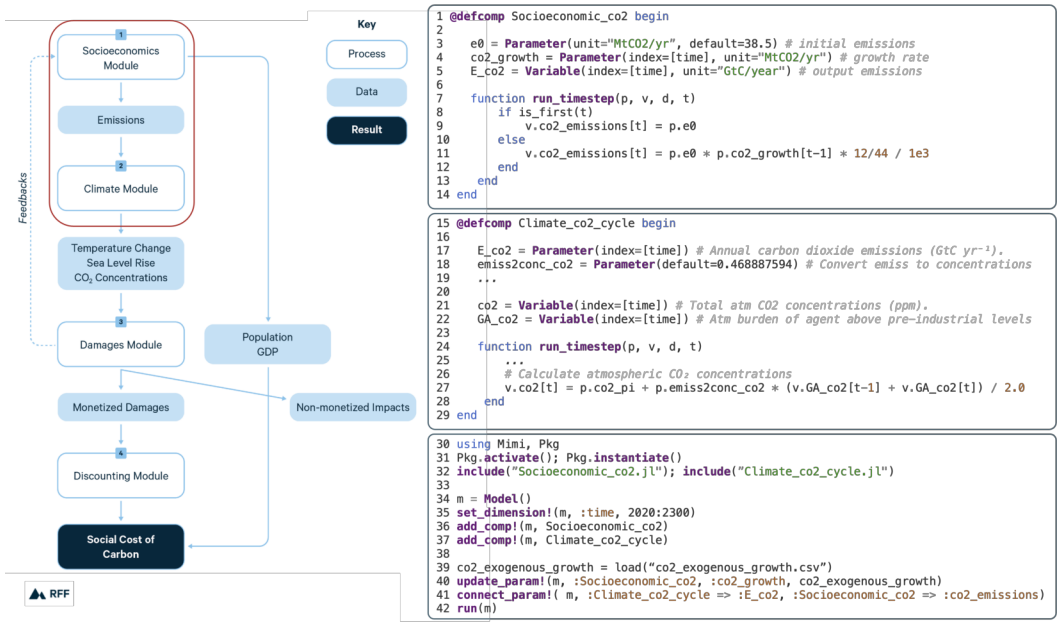

Fig. 1. A schematic of an IAM's structure [NASEM 2017; Rennert et al. 2022b] (left) and representative code snippets in Mimi (right).

The Mimi eDSL [Anthoff et al. 2023a] aims to meet this need for an integrated software tool that co-locates a range of existing and in-development IAMs. Mimi is a domain-specific language

embedded within the general-purpose Julia language. The design prioritizes closeness of mapping between domain-specific concepts and eDSL constructs. It handles much of the boilerplate code each user would otherwise have to re-implement, and focuses on a flow that mirrors the layout of the domain's scientific literature. It introduces a custom set of abstractions to streamline model creation and editing, facilitates collaboration across teams with a modular approach, and provides infrastructure for sensitivity analysis and results visualization.

As shown in Figure 1, IAMs are composed of modules (e.g. Socioeconomic, Climate, Damages, Discounting), each of which is composed of one or more `Components`—essentially, sub-modules (e.g., `Socioeconomic_co2` and `Climate_co2_cycle`). Each `Component` uses both exogenous `Parameters` from data files and endogenous `Parameters` from upstream `Components` to make calculations. Mimi's implementation leverages a range of Julia features including its support for macros, multiple dispatch, dynamic typing, high performance numerical computing, and Julia's environment and package management support [Bezanson et al. 2017]. Section 4.2.3 further ellaborates on these features in the context of Mimi user experience.

Figure 1 includes concrete snippets of a Mimi program, which we use to illustrate a few key Mimi features:

- To construct a model, the programmer sets dimensions (`set_dimension!` on line 35), adds `Components` (`add_comp!` on line numbers 36-37), and adds connections between them (`connect_param!` on line 41).
- Building and running models requires using the Julia package management system (`Pkg` functions) (30-31), and new models are often developed as Packages themselves.
- `Component`, `Parameter`, `Variable`, and `Model` are custom types which take advantage of Julia's support for multiple dispatch (see Section 4.2.3). For example, `run(m)` on line 42 calls the Julia `Base.run` function on Mimi's `Model` type. Mimi creators implemented a `Method` that defines the behavior when `Base.run` dispatches on a `Model`.
- Mimi uses macros to implement `Components` (e.g., `@defcomp` on lines 1 and 15), and in other contexts to implement Monte Carlo Simulations (`@defsim`). The syntax of `Components` and `SimulationDefinitions` is designed to closely resemble the host language's syntax for functions (see Section 4.2.2).

Mimi was designed with the goal of (i) promoting collaboration both within the academic community, and more broadly across the full range of stakeholders involved in climate policy-making, and (ii) reducing barriers to novel scientific inquiry [Anthoff et al. 2019, 2023a]. Recent publications suggest success in these goals, with Mimi being used both in academia [Bastien-Olvera and Moore 2021; Dietz et al. 2021; Errickson et al. 2021; Rennert et al. 2022a] and in policy [US Environmental Protection Agency 2022a,b].

3 STUDY DESIGN

We conducted a contextual inquiry [Holtzblatt and Beyer 1997; Holzblatt and Beyer 2017] study with nine participants, primarily from academic backgrounds with some representation from industry, the public sector, and NGOs. All participants used the eDSL under study, Mimi, in their current work. The study design is informed by existing literature and practices in ethnographic research, specifically as applied to the computing domain and the field of human computer interaction (HCI) [Crabtree et al. 2012; Goodwin 2015; Hacking 1991; Randall et al. 2007; Verne and Bratteteig 2018]. *Contextual inquiry* is a heavyweight methodology for collecting qualitative data about human participants' existing work practices. It emphasizes observing participants doing their own work in their own places of work (their *contexts*). During sessions, the researcher acts as a kind of apprentice; their role is to learn to do the work in the same way that the participant does the work—even if the

Table 1. Key information about study participants.

ID	Age	Employment	Yrs in Climate Change	Yrs of Programming	Yrs of Mimi Programming	Setting for Mimi.jl Work	Programming Education (formal & informal)	PLs and Tools Used in Last Two Weeks
P1	30	PhD Candidate, Geography	7	11	2	academic research	- Numerical modeling courses (BSc)	R
P2	32	Post-Doctoral Researcher, School of Government	10	10	5	academic research, government, and private sector	- MSc in Science and Executive Engineering - Informal training with Mimi developers	Julia
P3	28	Assistant Professor, Civil and Environmental Engineering	9	10	1	academic research	- 1 course in undergraduate - Many years of informal self study - Various online tutorials and parts of online classes	Python, Julia, Jupyter notebooks
P4	37	Economist with federal government	2	10	< 1	academic research and government	- Informal as a necessity to for research needs and interests	R, Stata, Julia, LaTeX, Python
P5	40	Assistant Professor, School of Marine Science and Policy	8	30	6	academic research	- Formal education in C++, Scheme	R, Python, Julia, Javascript
P6	59	Associate Professor, Department of Computational Data Sciences	30	40	3	academic research	- Formal education in Fortran and NetLogo - Self-taught Pascal, SAS, Julia, Python, R, Visual Basic	NetLogo, Julia
P7	22	Research Assistant for federal government	1	4	< 1	academic research	- Government - Statistics undergraduate degree (R) - Online courses (Python)	R, Julia, Python
P8	28	Research Associate at NGO	1	7	< 1	NGO	- Data science course (graduate school) - R-based ecological dynamics course	R, Julia
P9	33	Assistant Professor, School of Mathematical Sciences	11	13	2	academic research	- Introductory computer science (undergraduate) - Learning on the job and teaching Computer/Data Science	Julia, Python, R, Excel, GitHub, Google Colab, Jupyter notebooks

researcher already knows alternative ways of doing it. Contextual inquiry is especially useful for need-finding research, which focuses on identifying problems in existing work practices.

We designed our study around two research questions:

- What kinds of programming challenges do domain experts face during use of an eDSL designed for their domain?
- Which of these challenges arise specifically because the DSL is *embedded*, or because of interactions between the DSL and the host language?

Participants and Recruitment. We recruited nine participants via snowball sampling, starting with recruitment emails to individuals, research labs, and policy groups known to use the eDSL under study, as well as a recruitment post on the public Mimi user forum [Anthoff et al. 2019]. Recruitment used a screening survey to collect metadata and ensure that our sample of participants varied across several axes—background discipline or industry, years of experience in the climate

change domain, and levels of formal and informal computing education, and years of experience with programming and specifically with Mimi programming. See Table 1 for participant details. We recruited for diversity in order to avoid centering our analysis on experiences specific to a particular subpopulation of Mimi users. All participants were existing Mimi users, and we observed them working on their own Mimi-related projects during our contextual inquiry sessions. Thus, our work does not surface challenges specific to users who are *new* to a particular eDSL. As shown in Table 1 and confirmed during sessions, most participants are not *formally trained* programmers, and while they vary in years of experience and relative comfort with programming, they primarily use programming as a means for domain research and professional tasks.

Consent and Compensation. For our informed consent process, participants signed a consent form prior to beginning the study session. We compensated participants with a $40 gift card to either Amazon or a local business of their choice.

Session Protocol. Each session began with a short introduction to the study and verbal reaffirmation of informed consent. We then conducted an approximately 45-minute observation followed by a 5–10-minute semi-structured interview. In the observation period, participants worked on a task of their choice using the Mimi eDSL. All participants brought existing in-progress Mimi projects, or recently completed projects they hoped to turn back to in the near-future. During observation, in accordance with the contextual inquiry approach, we interrupted with questions when we needed clarifications about participant actions. In a semi-structured interview [Knott et al. 2022; Wood 1997], the researcher holds a conversation with a participant, guided by a predetermined set of open-ended questions but with the flexibility to explore responses or other directions freely. For example, our interview guide included prompts like:

> "Thinking back on your uses of Mimi in the last two weeks, can you list some of the specific tasks you carried out with as much detail as possible."
>
> "Thinking back to the last time you were frustrated with a Mimi functionality, or lack thereof, can you describe the experience? Furthermore, how did you try to figure out how to solve the problem?"

We also used much of the interview time to validate interpretations of participants' actions during the observation portion.

We conducted sessions remotely over Zoom and recorded them for subsequent analysis; recordings included video of the participant's screen, video of the participant, and audio of the conversation. The first author conducted all contextual inquiry sessions.

In keeping with the standard practices of contextual inquiry, participants (rather than researchers) chose the programming tasks we observed [Holtzblatt and Beyer 1997; Holzblatt and Beyer 2017]. This contextual inquiry approach offers a number of key benefits, of which we highlight two: (i) **Observing how participants use the eDSL, not how researchers *think* they use it.** While a fixed, researcher-designed task has some benefits for comparability across participants and more standardized quantitative results, it risks imposing researcher assumptions on the participants. Instead, contextual inquiry aims to reveal what participants do in their real work; this focuses our subsequent analysis on problems and user experiences that participants face in practice. (ii) **Wide range of use cases and workflows.** In addition to eliciting *realistic* needs, we aimed to elicit *diverse* needs. Enforcing a fixed task would have narrowed the range of observed problems to only the problems that can arise within the fixed task. As described above and reflected in Table 1, participants exhibited a range of backgrounds, use cases, and choices around tools and computing environments. This allowed us to observe a broader array of behaviors and challenges. Overall, study designs that allow participants to work within their usual environments and on familiar tasks have the potential to improve ecological validity (observations are more faithful to the real-world

user experience than unfamiliar lab-assigned environments and tasks) and breadth (observations vary more from participant to participant).

Note that while this is a user study, it is not an experiment. There is no comparison of a control group versus an intervention. Contextual inquiry is useful for our need-finding purposes because it allows us to observe a broad array of behaviors, challenges, and tasks; that same diversity means that comparison across participants is infeasible. More importantly, comparison across participants is *not a goal of this work*. Rather, the goal is to learn about as many programmer challenges as we can. See Section 6 for a deeper discussion of the kinds of research questions we can answer with contextual inquiry versus the kinds that require other types of data.

Although contextual inquiry is resource-intensive and produces data that is not amenable to shallow data analysis techniques, it offers rich, detailed information about real-world use cases [Holtzblatt and Beyer 1997; Holzblatt and Beyer 2017].

Data Analysis. We conducted an inductive, semantic thematic analysis [Braun and Clarke 2006] of the audiovisual recordings of study sessions using MaxQDA [Software 2022]. This analysis covered nine sessions and over 400 minutes of recordings. First, the first author used open coding to assign semantic codes and accompanying descriptions to short video segments (5-30 seconds each). In this phase, we aimed to associate purely descriptive text, without interpretation. Both authors reviewed the open codes that resulted from this phase. Next we began an iterative process of grouping these codes to produce the next layer in a hierarchy of axial codes. We added layers to the hierarchy as needed, consolidated and partitioned axial codes as needed, and refined code definitions throughout the iterative thematic analysis process. This culminated in several high-level themes, the top level of the axial code hierarchy, and these themes guide the structure of this paper.

4 FINDINGS

This section presents key findings, organized into five high-level themes: (i) Impacts of the host language: For example, the choice of host language affects access to other interoperable software, exposes participants to host-language error messages, and may shape their experience with package management systems. (ii) Impacts of the interaction between host and embedded language: For example, blurring the line between eDSL and host language behaviors can make tasks like debugging difficult for participants, syntax similarities between host languages and eDSLs led participants to make wrong guesses about allowable eDSL syntax, and eDSL implementation choices shape the host language features that participants must learn. (iii) EDSL-specific communities: Participants show a preference for eDSL-specific communities, and some relate adoption or continued use of the eDSL to the availability of these resources. (iv) EDSL-specific resources: Participants exhibit a heavy reliance on eDSL-specific tutorials and existing eDSL codebases, with implications for programming pattern replication over time and the importance of creating relevant reference material. (v) Integration of multiple tools via the host language: Participants necessarily integrate multiple software tools in their workflows, and are sensitive to the complexities and error-proneness of transferring data between tools.

4.1 Impacts of the Host Language

The choice of host language affects not only the work of the embedded DSL developers but also the *usability of the resultant DSL*. This is a dominant theme of this study: "My frustration with Julia and my frustration with Mimi run together a lot" (P8; 7 yrs programming, <1 yr Mimi). While the literature emphasizes that choice of host GPL has consequences for eDSL developers [Freeman and Pryce 2006; Kamin 1998; Mernik et al. 2005], this section focuses on the implication of host language choice *on eDSL users*.

4.1.1 Availability and Stability of Other Tools for the Host Language. The participant experience was substantially affected by (i) the availability and stability of Julia packages and (ii) the fact that writing Julia programs let them work within a large, well-supported ecosystem of tools and infrastructure, such as popular IDEs and linters. (See Section 7.1.1 for related literature.)

Participants reported struggling in cases where other Julia packages, other than the eDSL under study, changed rapidly or failed to suit their use cases. Julia is a fairly young language, and many packages are under rapid development. P2 expressed frustration with the need to rewrite their code to match syntax when packages released new (breaking) changes:

> "One of the things that has been annoying is that with Julia's updates, like reading the files for instance, the syntax has changed a bit ...and so this is some code that was started to be written maybe a couple of years ago, and so I've had to rewrite. I mean it's not a huge effort, but it definitely takes like half day or something to move it over." (P2; 10 yrs programming, 5 yrs Mimi)

Several participants expressed a desire for richer plotting packages in Julia. P2 used the Julia VegaLite package for plotting, but stated that "Julia is pretty new, VegaLite is pretty new, so there are some limitations" (P2). P1, P4, and P9 explicitly stepped out of Julia and used R's ggplot [Wickham 2016] to do plotting. This incurred the cost of data I/O, described in detail in Section 4.5. Because of this lack of sufficient Julia plotting support, P9's workflow involved flipping rapidly between Mimi and Julia in the Visual Studio Code (VSCode) IDE to run and edit scripts, and Python and Jupyter Notebooks [Jupyter 2022] for data exploration and plotting. This seemed to slow workflow, specifically disrupting file organization as P9 created disparate files that were difficult to track (again see Section 4.5).

In contrast, the availability of useful Julia packages substantially enriched participants' experiences with Mimi. P5, an advanced Mimi user with 30 years of programming experience and 6 years of Mimi experience, took advantage of Julia packages for optimization and linear programming. They said this integration had been "working quite well" (P5) for them. Leveraging the host language's strengths in numerical programming and performance was key to making Mimi useful for their work. Both P4 and P9 similarly took advantage of Julia's rich ecosystem of statistical packages, using them to add efficient and powerful statistical and optimization functionality.

Apart from packages, compatible programming tools and environments affected user experience. Most participants utilized an IDE like Juno [Juno 2022] or Visual Studio Code (VSCode) [Microsoft 2022] to do their work in addition to either integrated Github functionality in the IDE, or a Github Desktop Application. The host language, Julia, is well integrated into these tools, and Mimi piggybacks on this integration. For example, P3 took advantage of the fact that VSCode looks for environment configuration files and automatically instantiates the predefined environment when a Julia REPL is started. They said "I like that the VSCode extension will do that by default" (P3). P2 relied heavily on the fact that Julia's Vegalite [Anthoff 2022] package is integrated into VSCode such that plots pop up in an integrated window, and P9 used Jupyter Notebooks for diagnostics and other subtasks.

On the other hand, barriers to integrating tools with the host language can also be barriers to integrating those tools with the embedded language. By default, a Julia package is stored in a *hidden* folder, making it undiscoverable to default Jupyter Notebook instances. Setting up a Mimi model as a package thus makes it undiscoverable to Jupyter. P3: "you can't put a Jupyter notebook in a hidden folder and that's where I just quit" (P3). When embedded DSLs inherit obstacles to tool integration from their host languages, this can affect user experience.

Key Insight. Participant experience was affected not only by intrinsic features of the host language, but also by (i) the availability of other packages and eDSLs implemented for the host language and (ii) the availability of infrastructure and programming environments that support the host language.

Since embedded DSLs operate not as standalone languages, but members of an ecosystem of packages and tools, the usability of an eDSL is tied to (i) the host language's available tooling, (ii) integration with other software products, and (iii) the package ecosystem. If one is choosing the embedded approach in part to leverage this, it may be helpful for designers to take a holistic look at the how their DSL will fit and how compatible tooling will affect the user experience.

4.1.2 Host Language Error Messages. Error messages written by the Mimi developers use the concepts and vocabulary of the embedded DSL and are intended to be easily interpretable for a Mimi user. Errors *from Julia* often used unfamiliar concepts from the host language, which hindered debugging efforts. This finding echoes speculation from the literature that eDSL error messages may be more confusing than non-embedded DSL error message because they reflect the concepts and vocabulary of the host language instead of the DSL (Section 7.1.1).

Julia errors commonly refer to the Julia type system. If Julia cannot determine which function method to call based on the argument's run-time types, it throws a `MethodError` signaling that there is no `Method` for the given (argument) Type(s). For participants new to the Julia type system, these errors may seem unintelligible: P8 (7 yrs programming, <1 yr Mimi): "The Mimi API *errors in a similar way to the way Julia errors*, such that the error message for someone who isn't savvy in all the types like `AbstractBool` can be a little bit like morse code sometimes." (emphasis added)

Another example: Mimi extends the Julia `Base` function `getindex` such that one can call it on a `Mimi.Model` to obtain Parameter and Variable values. P7 (4 yrs programming, <1 yr Mimi) tried to access the parameter values of `paramname` of component `compname` in a model `mymodel` that they had not yet run; they typed `values = mymodel[:compname, :paramname]`. Because brackets are syntactic sugar for the `getindex` function, they encountered the error:

```
MethodError: no method matching getindex(::Nothing, ::Symbol, ::Symbol)
```

They noticed that the error message "said something about getindex" (P7), and knew this related to brackets, so they identified various places where they used brackets, but it took several minutes to match the error message's listing of the `(::Nothing, ::Symbol, ::Symbol)` types to the `mymodel`, `compname`, and `paramname` arguments respectively. Even when they managed to diagnose the issue, their ability to identify the error was largely influenced by "seeing this error before" (P7). For participants who were not comfortable with the Julia type system and the errors associated with it, this common category of `MethodError` was difficult for participants to understand.

In another case, an error message in response to P3's small syntax error (including an additional semicolon) used language specific to Julia metaprogramming:

```
... LoadError: Unknown argument name: '$Expr[: parameters, :($Expr(:kw, :kindex, :[time ...
```

This error took the participant about ten minutes to resolve. (See further detail in 4.2.2.)

Key Insight. Error messages that rely on users understanding host-language vocabulary degraded participants' user experience, especially for those new to the host language.

Participants' ability to understand the error messages they encountered, and to debug their programs accordingly, was crucial for user experience. While Mimi developers may aim to provide custom error messages that use constructs and vocabulary consistent with the eDSL, shielding users from the host errors entirely is difficult. However, eDSL design decisions affect which classes of host-language errors become common. For example, the design decision to use Julia macros to implement Mimi made macro-related Julia errors more likely. Overall, where Mimi implementation decisions resulted in users seeing unfamiliar host-language concepts in error messages, debugging was time-consuming and difficult.

4.1.3 Host Language Package Management System. For eDSLs implemented as packages, selecting a host language also means selecting the package management system that users must understand. Since Mimi is embedded in Julia, participants had to learn the Julia package management system.

P3 stated that they "love the Julia environments" (P3), though they took some effort to learn. P7 emphasized the learning curve as well:

> "When I first started using Mimi, it was quite difficult and frustrating because I didn't understand how all the dependencies and environments and all that kind of worked. But as I've gotten better with Mimi things have gotten generally less frustrating." (P7; 4 yrs programming, <1 yr Mimi)

During their session, P4 (10 yrs programming, <1 yr Mimi) seamlessly and frequently used `Pkg> st` in the REPL to check what packages, and what versions, were in their active environment. They said of this check that "these are just things I do because things happen when I'm not paying attention, and I do not know why" (P4). They also integrated the `Pkg` package into their scripts' preamble, including:

```
# Activate the project and make sure all packages are installed.
Pkg.activate("source_folder_name")
Pkg.instantiate()
```

This ensured the intended environment was activated in their working session and that all necessary packages were downloaded. P9 did the same. This preamble may indicate these participants have developed guardrails in response to prior difficulties from forgetting to update the Julia environment state.

P1, P2, and P6 used package management to add and use packages without observed trouble, even when P6 saw an error indicating the need to download a specific package, a problem they diagnosed quickly. Participants did perceive package installation as time consuming and tedious. P8 started a fresh project with the default (empty) environment. When they realized that they had a pre-existing environment that would work for their experiment, they switched to that one to avoid the time and the verbose process of re-adding packages to a new environment.

Overall, levels of comfort with Julia's package management varied. Some participants seemed to have smoothed the process by erecting guardrails for their own processes, updating packages automatically to avoid some common pitfalls, while others continued to manage them manually. The visibility into environment state, verbosity, and slow precompilation time of Julia package management are common complaints in the Julia community [Boudreau 2019], and we saw these concerns echoed by Mimi users.

Although we intended our study design primarily to reveal issues around the interaction of eDSLs with their host languages, this topic gives us an opening to discuss opportunities around the interactions of eDSLs with language tooling more broadly. In particular, note the common threads throughout the package management struggles described above: *visibility* into the package environment and *variations* in the package environment. Participants repeatedly using `Pkg>st` were struggling with visibility. Participants who added guardrails that ensured a particular fixed set of packages were struggling with variation. Participants talking about "things happen[ing] when I'm not paying attention" were struggling with both visibility and variation—and in fact the two are closely related. The fact that these were the core issues suggests a role for IDEs—or other language support tooling—that support users in maintaining awareness of current environment state or in freezing environment states. We speculate that any intervention that makes environment state more visible, harder to miss, or that makes it difficult to alter the environment in non-reproducible ways may ultimately make eDSLs more usable for the kinds of users we studied. For language designers or IDE designers working on package management support who want to support eDSL users of their language or tool, visibility of environment state should be a first-class concern.

Key Insight. For an eDSL implemented as a package, the usability of the package management system affected participants' user experience.

Since many eDSLs are implemented as packages for the host language, designers may wish to consider the usability of the package management system in selecting a host language. It may even be worth preemptively assessing the usability of multiple candidate host languages' package management systems with the target audience. Since package management is a known usability barrier within the Julia community, we also note a secondary key insight: Designers may wish to identify commonly-known host language usability issues in order to preemptively smooth out the experience for their eDSL users.

4.1.4 Impacts of Host Language Summary. The choice of a host language directly affects the experience of the end user via:

- Access to other, interoperable host language software—e.g., libraries or programming environments available for the host language—can affect the user experience of the eDSL.
- The eDSL implementation can expose users to error messages from the host language—and especially error messages that use host-language terminology rather than domain-specific terminology—that affect user experience.
- For eDSLs implemented as packages, the package management system is one component of the host language that eDSL users will definitely need to learn, so package management affects user experience.

Overall, the advantages of embedding a DSL in an existing host language, such as piggybacking on tooling and access to rich host language features, also come with risks of confusing users and burdening them with extra learning demands. This may be especially notable for users new to both the eDSL *and the host language*. Our findings suggest that designers may wish to consider how their implementation choices affect the three areas above. How will the eDSL implementation affect which other host-language software and tooling is trivially inter-operable vs. difficult to hook together? For common categories of errors, will the resultant error messages express issues in the vocabulary of the eDSL or the host language? Can the target audience be productive with the host language's package manager? And finally, are there known user experience barriers associated with the host language?

4.2 Impacts of the Interaction Between Host and Embedded Language

The findings so far emphasize aspects of the host language: availability of libraries and tools, error message quality, package manager quality. DSL implementation choices can help users benefit in the positive case (programming environment support) or help shelter users in the negative case (confusing error messages). However, these are first and foremost features of the host language, *not* features of the interaction between the host and eDSL. In this subsection, we turn our attention to interactions between the host and the eDSL.

4.2.1 Distinction Between Host Language and eDSL Behavior. Although many design guidelines suggest a "seamless" integration of the eDSL into the host language, we observed that participants often struggled when they could not find the seams.

Participants expressed confusion about the distinction between Mimi and Julia and sought clarity on the dividing lines. These patterns may cast doubt on the design tenet that embedded DSLs should be a "seamless extensions to the language" [Freeman and Pryce 2006]. Section 4.1.2 detailed how blurring the GPL–eDSL line damaged participants' understanding of error messages, but this same pattern appears in a variety of interactions.

We observed that participants often felt they needed to know where Julia ended and Mimi began. P1 was curious enough about the language differences to open the Mimi source code locally:

> "I was just opening the Julia files ... to get familiar with some Julia functions and what was Mimi and what was Julia because ... I was new to Julia so for me it was like I don't know what is something very specific from Mimi." (P1; 11 yrs programming, 2 yrs Mimi)

The blurred GPL–eDSL distinction especially hampered debugging. When participants mistook Mimi behaviors for Julia behaviors or Julia behaviors for Mimi behaviors, they expressed that they did not know where to look to debug. For example, when P3 encountered a Julia parsing error, they tried to use the Julia inline REPL `help?>` functionality to search `help?> Mimi.Parameter` and, when that did not work, `help?> Parameter`. Since the `help?>` functionality only supports Julia and not Mimi, they were unable to find any documentation.

A few users also wondered aloud whether certain pain points, such as performance issues, were a Mimi behavior, a Julia characteristic, or a personal computer hardware problem (P4, P7). For instance, "The explorer window [Mimi data visualization UI] takes a really long time to open, that's a bit annoying sometimes, I don't know if .. is that a Mimi thing?" (P4).

Key Insight. Blurring the line between eDSL and host language made some tasks, especially debugging tasks, ambiguous for participants; a blurred dividing line can leave it unclear whether to look for host or eDSL resources and whether to dig into host or eDSL code.

Although eDSL design guides often suggest blurring or completely disguising the line between the host and embedded language, this may sometimes stand in the way of user goals. For participants identifying where to search for help or tracking down the source of performance issues, knowing more about what was being handled by the host language versus the eDSL may have helped them identify next steps.

4.2.2 Distinction Between Host Language and eDSL Syntax. While the prior discussion centered on cases where participants wanted to know which behaviors to attribute to Julia vs. Mimi, this discussion covers cases where participants needed to distinguish Julia and Mimi *syntax*. Again, the literature recommends that "the EDSL syntax should make no distinction between the built-in features of the language and those provided by the [eDSL designer] to support their application" [Freeman and Pryce 2006]. And again, design choices that blurred the line between GPL and eDSL blocked progress.

Prior literature argues that in cases where the syntax matches exactly, these kinds of parallels may help users pick up eDSL syntax. However, in cases where the syntax is similar in some but not all respects, these kinds of parallels may produce difficult-to-diagnose issues. This dovetails with literature warning that matching syntax can constrain design and reduce clarity for domain users [Cavé et al. 2010; Mernik et al. 2005; Poltronieri et al. 2018b].

As an example of the dangers of looking too similar to host-language syntax, even programming tools may draw unwarranted parallels between host-language syntax and eDSL syntax. P3 (10 yrs programming, 1 yr Mimi) uncovered an unexpected syntax mismatch via using an automatic code formatter. Their automatic formatter put a semicolon between required arguments and those with default values. This converted `myfunc(2, option=false)` to `myfunc(2; option=false)`. This is standard recommended practice for formatting arguments to Julia functions, to increase readability.

This behavior caused problems for P3 when it interacted with Mimi syntax. Mimi's macro-implemented abstractions use syntax that resembles—but *does not exactly reproduce*—Julia function syntax. Some Mimi macros include elements that resemble Julia function syntax, but which are actually parsed in the macro instead of run directly as functions. The choice to use similar syntax partly

obscures that, under the surface, Mimi designers have implemented new, complex functionality. For example, the following Mimi snippet creates a `Parameter`:

```
temperature = Parameter(index=[time], unit="degC")
```

When P3 wrote this line, their formatter mistook it for a Julia function call, rather than an expression to be parsed within a Mimi macro, and automatically inserted a semicolon:

```
temperature = Parameter(; index=[time], unit="degC")
```

While the syntax of this macro expression closely resembles a Julia function call, the parsing logic in the macro implementation does not handle semicolons, so adding a semicolon to the line caused an error. P3 observed a low-level parsing error including the text:

```
... LoadError: Unknown argument name: '$Expr[: parameters, :($Expr(:kw, :kindex, :[time ...
```

The participant spent ten minutes trying to solve this problem, seeking help from online documentation and commenting out lines to try to isolate the error. They were eventually able to compare to existing, functioning code and online examples to recognize that the extraneous semicolon did not match any examples or working code. Templates were useful for problem solving here (see Section 4.4 for more on templates), but the participant had trouble understanding exactly *why* the problem occurred. In this case, substantial similarities coupled with subtle inconsistency between eDSL syntax and host language syntax resulted in a low-level error message about parsing that did not match user concepts or knowledge and slowed progress significantly.

Overall, this theme naturally raises questions about implementing abstractions with macros. The DSL implementation literature suggests that the embedded approach constrains language designers [Cavé et al. 2010; Freeman and Pryce 2006; Mernik et al. 2005; Poltronieri et al. 2018b; van Deursen et al. 2000]; macros offer them the opportunity to regain some control, maybe even to implement abstractions that more closely match their target audience's mental models [Blackwell and Green 2003]. Although it is hardly a new finding that macros present trade offs for designers and can complicate the ability of designers and users alike to reason about code [Ballantyne et al. 2020; Becker et al. 2019; Brabrand and Schwartzbach 2002; Culpepper 2012; Culpepper and Felleisen 2004], and that macro-related error messages and debugging are already a known usability issue [Becker et al. 2019; Culpepper and Felleisen 2007; Dévai et al. 2015; Hemel et al. 2011; Niebler 2007], our findings suggest an additional class of situations in which macros are likely to degrade user experience: using macros to implement syntax that is *similar but not an exact match* with host-language syntax.

Key Insight. In cases where an eDSL implementation diverges from its host language, explicitly *differentiating* between host language and eDSL syntax, as opposed to attempting to mimic syntax and blur the distinction, may improve the user experience.

While making eDSL syntax similar to host language syntax may often help, looking too much like host language syntax may hinder users. Especially during debugging, if the syntax does not signal whether they are interacting with a host-language or eDSL construct, users may not know where to seek help or understand why their code is failing when it looks similar to working host-language code. Explicit differentiation may help in these cases.

4.2.3 *eDSL Users Must Learn Parts of the Host Language, and eDSL Design Controls Which Parts.*

When designers implement eDSLs, they make decisions about what parts of the host language users have to learn. Design advice often emphasizes that being able to use a variety of host language features makes eDSL creation easier *for designers* (Sec 7.1.1). However, asking users to understand more host language features can make the eDSL harder to learn and use *for those users*.

Participants identified a lack of Julia understanding as a key obstacle to their progress with Mimi:

> "It would be nice if there was some initial background on a few simple things on Julia, because [Mimi] is a package that works in Julia ... A few things will come up that are sort of ... maybe I should have learned Julia more first, but it's sort of presented as you don't necessarily need to." (P6; 40 yrs programming, 3 yrs Mimi)

Package Development. Mimi documentation encourages formatting completed Mimi Models as Julia *Packages* as opposed to *Projects* [Anthoff et al. 2023b,c]. This design decision means that a user working with one of these completed models must understand how *developing a Package* [Julia Contributors 2022a] differs from working on a Project in Julia, and the implications for workflow. Registering Mimi models as packages in the Julia General Registry integrates them into documented Julia workflows and improves user experience for domain experts using them in projects—but it can be a barrier for the users developing the packages themselves.

For over 6 months each, P4 and P7, both with less than one year of Mimi experience, developed existing Models-as-packages using unconventional workflows that diverge from what Julia documentation suggests for package development. They experienced continuous frustration with not seeing their changes to core package scripts picked up when using that package, and their strategies to get changes picked up were unreliable and hard to understand. This was caused primarily by their systems pointing to the packages held in the General Registry, but changes being made to the locally developed version and not synced to the General Registry. Julia's recommended package-style workflow would have simplified this problem, but participants continued to use a project-style workflow. This indicated they may not have been aware of how package development differs from other workflows.

> "One of the things that I think is most frustrating for me as a novice is why didn't my edits take and stuff? So I started out and used to go back all the way to `using` and have it re-precompile, just because there are so many hours and days where I've spent where my edits weren't accounted for. Obviously my fault because there was something wrong I was doing in my workflow." (P4; 10 yrs programming, <1 yr Mimi)

Eventually these participants learned, from experience in addition to the Mimi forum and developer assistance, about a more reliable and conventional workflow for package development and decided to convert their workflows. Even after shifting to the recommended workflow, they were wary of changes not being picked up, so much so that they begin each working session by killing the REPL: "One thing I do is always at least kill the REPL because that's been extremely unreliable as to what it's actually reading or is precompiled, and I haven't really figured that out." (P4).

In their session, P7 spent over 30 minutes trying to transition to the new suggested workflow. This included reading documentation, watching a video produced by the Mimi development team, deleting and re-downloading (or moving) local copies of in-development packages to the local system Julia *development* folder, opening those packages to clean up their branches and configurations, and finally rebuilding configuration files of projects to link to these local *development* copies. This process was slow and iterative, including working through error messages, returning to documentation, and dealing with additional complications caused by using unconventional workflows for so long.

Even once these participants transitioned to conventional workflows, interaction with package development remained a usability barrier. Getting changes reflected in a running system is reliable but slow using REPL restarts and precompilation. The `Revise` package allows many (but not all) changes to be incorporated without killing the REPL, and P4 and P7 eventually learned about it, but only by chance during offhand conversations with Mimi developers.

Julia package development is not a simple endeavor, as evidenced by a host of supplementary how-to guides [de Balsch 2021; DSB 2020], questions on forums, and even custom programming environment tooling in IDEs like VSCode [Anthoff 2020]. P3 even sought an additional package

called *DrWatson* to help them manage their projects and development. *The design decision to include package development in the Mimi user workflow meant that participants needed to understand this complicated process.*

Multiple Dispatch. The Mimi implementation makes heavy use of Julia's support for *multiple dispatch* [Conributors 2022; Julia Contributors 2022b]. See Section 2.2 for a discussion of how Mimi uses multiple dispatch, and see Figure 1 for example code. The use of multiple dispatch allows Mimi to essentially overload Julia Base functions like `getindex`, `show`, `build`, or external package functions `DataFrames.getdataframe` and `CSVFiles.load`, so that these operators can accept arguments with Mimi-specific types. This allows Mimi to maintain a relatively small code-base and take advantage of host functionality and performance, keeps syntax consistent between eDSL and GPL, and promotes rapid light development processes. That said, it also means that Mimi users are exposed to error messages associated with multiple dispatch and that they must develop debugging strategies for failures related to multiple dispatch.

For participants who describe themselves as not "savvy with types" (P8; 7 yrs programming, <1 yr Mimi), multiple dispatch errors—which rely on familiarity with the involved types and some understanding of how Julia dispatches based on types—represented a user experience pain point. Section 4.1.2 covers error messages and describes P6's experience with the `Symbol` type and P7's with the `Nothing` type. We also observed that both P4 and P7 interacted with the Julia `Nothing` type, both in error messages and in exploring object structures, since Mimi frequently uses the value `nothing` of type `Nothing` to stand in for unset parameters or attributes.

Although participants experienced type-related confusions outside of multiple dispatch situations, it is clear that *the choice to use multiple dispatch in Mimi's implementation had the knock on effect of exposing Mimi users to an apparently confusing category of errors that reference the overloading concept.*

Common Julia Issues the eDSL Successfully Avoided. Mimi's design did seem to successfully shield participants from some common Julia user experience issues. For example, the eDSL somewhat protected participants from performance pitfalls, especially in potentially low-performance tasks like Monte Carlo Simulations [Nissen 2022a]. Some programmers find Julia documentation thin [Bezanson 2019; Boudreau 2021], but we did not identify this as a primary challenge for our participants, who had access to the Mimi documentation and forum (Section 4.3). The Julia community identifies confusion and frustration around subtyping logic, method specificity rules, and the verbosity of Union type declarations as user experience issues [Bezanson 2019; Nissen 2022b]. However, we saw no evidence of this problem among participants, perhaps because the eDSL implements the core composite types (Model, Component, Variable, etc.) and their methods; thus participants were not required, nor apparently inclined, to engage with these Julia features. This design choice also shielded participants from the common complaint that Julia does not have good protocols for implementation of types that support common interfaces [Bezanson 2019]. *By avoiding some known-difficult host language features, Mimi may have prevented some potential user experience barriers.*

Summary.

Key Insight. Embedded DSL implementation choices affected which features of the host language participants had to learn. Enforcing that users must learn difficult host language functionality may affect the learnability of the eDSL or slow debugging efforts.

Overall, Mimi's design caused participants to interact with particular elements of Julia, including package authoring and multiple dispatch, and it was clear their interactions with those Julia elements drove some of their struggles with Mimi. Asking users to understand more host language

features can make the eDSL harder to learn, especially for users new to the host language. On the other hand, Mimi successfully shielded users from some common host language issues. This suggests that designing learnable, usable eDSLs may require understanding how implementation decisions affect the parts of the host language that users will be required to employ and reason about. This kind of understanding may help designers change implementations to shield users from known-complicated parts of the host language.

4.2.4 Impacts of Interactions Summary. Overall, interactions between the host language and the eDSL affected the experience of the end user via:

- Blurring the line between eDSL and host language behaviors can degrade user experience for users who are new to the host language, especially for debugging tasks.
- While making eDSL syntax and host language syntax look similar may help host-language experts make good guesses about eDSL syntax, a near-perfect match may sometimes hinder usability more than an explicit differentiation.
- The eDSL implementation can hide complex host language features or, either intentionally or unintentionally, require eDSL users to learn them.

As in Section 4.1, it is clear that embedding a DSL in an existing host language offers the advantage of piggybacking on access to rich host language features, but also risks confusing users and burdening them with extra learning demands. Also as in Section 4.1, the bad outcomes may be especially common for users new to both the eDSL *and the host language*, which may be more common for domain experts. We suggest designers may wish to consider how their implementation choices affect the three areas above. For common categories of errors, performance issues, and other failures, will users know whether to blame the eDSL or the host language, and will they know where to look for debugging help? For common syntax errors, will users know whether they are debugging a host language construct or an eDSL construct? Which host language features will users need to understand in order to be productive with the eDSL? And finally, are there known user experience barriers associated with some features of the host language, and can the eDSL design shelter users from those features?

4.3 Engaging with eDSL-Specific Communities

Participants sought out domain-specific language communities as opposed to engaging with host-language communities, and cited collaboration with the Mimi community as a driver of adoption.

4.3.1 An eDSL-Specific Community. Participants engaged readily with the Mimi community, but much more rarely with host language and language-agnostic communities. In searches of two common Julia resources—Stack Overflow and the Julia Language Discourse—we observed no evidence of Mimi users directing questions to the Julia community at large. In contrast, we find *eight of nine participants* engaging on the Mimi forum [Anthoff et al. 2019].

Of the ability to contact the Mimi community with questions, P2 stated that "It has been absolutely crucial ... I could lose days on this and instead I just write [the Mimi developers], and in a few hours I have an answer. It's incredible. My friends are jealous" (P2). Similarly, P1 said that their Mimi public forum entries held a record of their toughest challenges, because that is where they turned when they needed help beyond documentation. They said they avoided posting questions to StackOverflow, assuming "it's going to take years" (P1) to get a response. In contrast, they said the Mimi forum "was the first and only time I asked ... actually typing a question and asking for help ... [it was] cool that I got responses and saw what I wanted to do" (P1).

Key Insight. Participants preferred forums and communities that cater specifically to the eDSL, rather than to the host language.

Our observations suggest eDSL users may prefer eDSL-specific forums and interaction with an active eDSL user community for satisfactory user experience. These observations may be shaped by a number of cultural factors—e.g., prior exposure to programming, domains of expertise—but the reliance on Mimi-specific communities and overall underuse of Julia-associated or language-agnostic resources suggests that at least for some audiences, an eDSL's user experience may be affected by access to their own specialized communities.

4.3.2 Community Engagement Impact on Adoption of eDSLs. Participants also reported that community engagement was key for adoption. This pattern is known for GPLs [Head 2016; Meyerovich and Rabkin 2013], but our participants suggest the same pattern may apply for eDSLs.

Many of the study participants either began using Mimi because their research team or colleagues use Mimi, or so they could work specifically with new groups or individuals of interest. P1 attended a workshop for about 50 domain experts and found that the "workshop sold it well" (P1) by presenting not only how to use the DSL, but also the collaboration opportunities:

> "I guess that also seeing other papers using it and, like, having the thought that at some point I could kind of implement or, like, merge these components was tempting ... It was like 'Ok so this is a way to get close to research groups that are using it'." (P1; 11 yrs programming, 2 yrs Mimi)

Most participants worked on augmenting or modifying existing models in Mimi as opposed to starting a brand-new model or project from scratch. We discuss this pattern in more detail in Section 4.4. Participants reported that this kind of asynchronous collaboration with the Mimi community was also a key driver of adoption.

Key Insight. Participants reported that eDSL-specific support and seeing a thriving eDSL community played a role in their eDSL adoption decisions.

Since our contextual inquiry only observed existing users of Mimi, our data did not include *observations* of eDSL adoption decisions. However, participants self-reported that learning about the Mimi community—and having access to it—influenced their choice to adopt Mimi themselves.

4.4 Engaging with eDSL-Specific Tutorials and Codebases

Participants used Mimi tutorials and fellow Mimi users' programs as starting points for their work. Section 4.3 highlighted participants' engagement with domain-specific user communities. Here we make the related observation that participants looked to code from either Mimi tutorials, or other Mimi users, as a primary source of templates. Participants frequently copy-and-pasted scripts from these sources to begin their work. Most did not start from scratch but incrementally modified an existing publicly available Mimi Model. This behavior seemed mostly productive, given Mimi support for Model modification and community engagement, although in some cases it could lead to bloated or confusing programs.

4.4.1 Copy-Paste-Modify. Many participants started a new program by finding whole programs or snippets of existing programs that they could adapt to meet their needs.

For example, P4 made several incremental modifications to an existing model. For each modification, they (i) looked through the existing model for examples of the function or syntax they wanted to use, (ii) directly copy-and-pasted the example code to their desired location and, (iii) modified as necessary. When they encountered errors, they went back to the copied examples and compared to their modified version to understand what delta might have caused a bug—effectively using the original code as an assumed-correct template. P1, P2, and P7 followed the same pattern. P1:

> "What I do, or what I did here, was basically to retype again the models or the components ... I went to main components and just opened them in a Notepad document, and just the ones that I wanted to change are the ones that I retyped here. I just kind of copy-and-pasted what was within each component. I added the new parameters that I wanted." (P1; 11 yrs programming, 2 yrs Mimi)

Key Insight. Rather than starting new programs from scratch, participants started from an existing eDSL program and tweaked it to meet their needs.

If a new eDSL's users are likely to work in this same way, tweaking a starter program to meet their needs, it may be beneficial for eDSL designs to include abstractions designed to make this incremental modification behavior easier for users. Mimi's modification functions and modular structure seems to successfully achieve this goal. The fact that Mimi abstractions support this behavior may have encouraged this practice; however, existing literature on blank page syndrome and the use of templates suggests this behavior is prevalent across a variety of domains [Bloch 2006; Chasins 2019; Felleisen et al. 2018; Krishnamurthi and Nelson 2019; Nosál' et al. 2017; Thayer et al. 2021]. This copy-paste-modify behavior also means that practices exhibited in early public codebases may linger and propagate through the community for a long time. Designers may wish to publish sample programs based on the understanding that the structure of the sample programs will be replicated both by current and future users.

4.4.2 Tutorials as Templates. Many participants used Mimi tutorials in particular as a source of starter code. The use of code examples and copy/pasting of templates (also see Section 4.4.1) is an important phenomenon studied in several prior works [Head et al. 2018; Robillard 2009; Robillard and DeLine 2011; Sacks 1994].

For example, P3 wanted to run a Monte Carlo Simulation (MCS) on their model, so they looked to the documentation online, found a tutorial on MCS with Mimi, and directly copy-and-pasted the tutorial into their project before making minimal alterations. While the result was functional, it was bloated with unnecessary code. The tutorial was an "everything-but-the-kitchen-sink" tutorial, including many possible choices within one example instead of a concise representation of what a given user might want for a single task. For example, it presented several ways of carrying out the same task within one script, such that some created variables were unnecessary and unused. P3 directly copied the tutorial, *and thus included the unused code.* They paused and expressed confusion as to why the first line was included, reviewing their work closely. Similarly, P6 worked through the MCS tutorial during their session, and commented directly on this script "So actually it seems a little odd that ... I'm never actually using" (P6) the random variable `RV1`.

When desired programs diverged substantially from available templates, participants' use of templates resulted in confusing program designs. For example, P2 added a component to a model originally built of several short components. The new component had similar *conceptual* scope, but far more detail in its representation of processes. P2 had to frequently scroll through the component to build understanding, and they frequently paused to search for a specific section of interest. This program might have been more manageable and readable if it had been broken into subcomponents rather than emulating the template's structure. While P2 noted that the component was long, even inconveniently long, they did not seem to consider breaking it up into subcomponents.

Overall, closely mimicking template structures led to a variety of participant frustrations. P7 augmented an existing Integrated Assessment Model (IAM) by adding the ability to calculate the social cost of hydrofluorocarbon. They first found the portion of the model codebase that handled the existing set of gas options in an `if-elseif-else` conditional: `:CO2` (carbon dioxide), `:N2O` (nitrous oxide), and `:CH4` (methane). The blocks associated with each gas were nearly identical. As a first attempt, P7 (4 yrs programming, <1 yr Mimi) simply added a new condition to the existing

conditional (`elseif gas == :HFC`) and duplicated the block associated with another gas. While they "originally tried to just copy this [existing] style of adding a component" (P7), they soon realized that the data they needed for the new computation was stored differently than for the other gases, and thus the new computation diverged more than expected from the existing structure. Trying to mirror the existing code while adapting it for a new computation forced the participant to grapple with internal implementation details outside of the conventionally used public API. The participant expressed discomfort and hesitance after implementing the changes and said they hoped the adaptation "doesn't mess anything else up" (P7) and that it "*might* be an OK approach" (P7). We saw similar behavior from P4, who mentioned a syntactically simpler approach to expressing their program but picked the approach that matched the template they had used as a starting point "just because I do not want to mess things up."

Key Insight. Participants sought eDSL-specific documentation and tutorials, and they used the presented sample programs as templates.

Heavy use of eDSL-specific tutorials as templates suggests two key lessons for eDSL designers: (i) The embedded approach lets designers circumvent many of the implementation burdens associated with creating a GPL, but it may not exempt them from creating full-scale documentation—that is, documentation that supports users in end-to-end tasks, rather than only in understanding how the eDSL extends the host language. (ii) The fact that tutorial code will be used not only as a learning resource but also as *templates* for user code suggests additional constraints on the design of eDSL documentation and tutorials.

4.5 Integrating Multiple Tools via Host Language

Many participants reported benefiting from being able to integrate multiple tools, all within Julia. Climate economics scripts typically involve a patchwork of many tools and painstaking data I/O to transition data between them. In contrast to Section 4.1.1 which centered on the specific other software available in a given language, this thread of our analysis revolves around the fact that by simply being embedded within a host language, an eDSL has the potential to help users avoid the frustrating and tedious process of writing data out of one tool and reading it into another.

Modeling in the climate economics domain frequently involves the input, processing, and output of tabular data. For example, P3's session was largely taken up by the participant creating and testing small data I/O functions to read data from disk and manipulate them into a preferred format. This process was iterative, including viewing files within the IDE and running snippets of code to check outputs in the REPL. Since Julia has extensive data processing tooling and functionality, Mimi offers extensive support for these tasks with minimal Mimi-specific implementation effort. In contrast, a free-standing DSL would have to implement data I/O support from scratch.

Mimi represents integrated assessment models with a modular structure, using self-contained `Component`s that are then linked together to build a `Model` with a few simple functions like `connect_parameter!` that construct links and dictate the flow of data between components. Users link combinations of existing and original components to build a model, then direct original or modified datasets through these components, *without ever needing to write data out to files or read it back in*. Before the introduction of Mimi, the disconnected landscape of models, languages, and frameworks meant that users typically had to read and write data to disk, often passing it *between different programming languages and file formats*, to carry out the same work.

Both P7 and P2 explicitly describe the data I/O advantage of the Mimi approach. P7 said of components from different models:

> "It was cool to see them working together like that. You could just add the components from one model to another model and have them all run together. It's satisfying, I think. I think before, outside of Mimi and

pre-Mimi, when they were doing a lot of integrating models together work it *involved a lot of exporting CSVs of outputs from one model and then importing it to another and stuff like that, which is really slow and a bit clunky.* So it's nice to be able to do things ... that will just pull the inputs in directly *rather than needing to export CSV of outputs and pull it back in.*" (P7, emphasis added; 4 yrs programming, <1 yr Mimi)

P2 said that one of their top priorities was to avoid language switching, especially in this work where doing so includes a lot of data transfer:

"Going back and forth between languages, to me ... Like, that's my top priority is to try and avoid doing that. Because I think it's just too time consuming, and I know myself and, like, manipulating datasets or whatever it is, from one thing to another, I'm going to make mistakes. To me it's similar to just doing stuff in Excel. You just make mistakes, and you do not see them. So I prefer keeping everything in one language." (P2; 10 yrs programming, 5 yrs Mimi)

Key Insight. Participants expressed that the embedded approach reduced the unpleasant, tedious, and error-prone work associated with transitioning data using a patchwork of free-standing domain-specific tools.

The pain of transitioning data to and from disk, transitioning it between file formats, and transitioning it between languages makes embedding an attractive DSL implementation approach. With a free-standing DSL, the DSL becomes one more tool in the user's forest of tools, one more data input format and data output format to learn. In contrast, Mimi gave participants access to the whole range of Julia packages for doing their work within a single host language and without addition additional stages of reading and writing data on disk. Overall, eDSLs may reduce the volume of manual and error-prone data I/O operations that domain experts find unpleasant.

5 DESIGN IMPLICATIONS AND DISCUSSION

Below we summarize core lessons of our work that designers can consider as they design future eDSLs. We follow with a discussion of design implications and areas for future research.

5.1 Design Considerations

5.1.1 *Properties of the host language affect eDSL users, not just eDSL developers.*

The user experience associated with an embedded DSL is strongly influenced by other available packages and libraries and the ease of integration with other programming tools and environments. When choosing the host language for an eDSL, designers may wish to consider what functionality—and what problems—the eDSL will inherit from other libraries, and if the host language works well with the tooling and programming environments that users will need. The fact that "features come automatically and for free" [Kamin 1998] from a host language is a key *advantage* of the embedded approach. However, for host languages saddled with difficult library ecosystems or difficult tooling, choosing that host language will saddle eDSL users with the same issues. Designers should consider weighing *which* features in particular come with a given host language and choosing a host based on an understanding of what features their target audience will value most.

Errors expressed in host language concepts may slow down and confuse users. Designers should be aware of common cases in which users may be exposed to host language concepts in error messages. This finding provides empirical backing for existing claims in the literature [Freeman and Pryce 2006; Kamin 1998; Mernik et al. 2005]. Designers may wish to consider catching and customizing these messages as much as possible. Especially for domain experts, designers should not assume that users will have a deep understanding of the host language beyond what they are exposed to in the embedded language, and this should shape error message design.

When an embedded DSL designer chooses a host language, they also choose the package management system programmers must use. DSL designers may choose an embedded approach partially so that users will be able to take advantage of other libraries already implemented in the host language. Users who attempt to do so will be required to use the host language's package management system, which may or may not be intuitive for them.

Blurring the distinction between host language and embedded language may confuse and hinder participants. Seamless integration of the embedded DSL into the host language is a popular design guideline, and it may have advantages for both user and developer [Freeman and Pryce 2006; Karsai et al. 2014]. That said, our findings indicate that designers should anticipate that confusion about the difference between the host language and the eDSL may stand in the way of users' goals, especially during debugging interactions. This may be especially prevalent if users are new to *both* the host and the domain language. In some scenarios, a clear distinction between host language and eDSL, as opposed to forcing a near-perfect match, may help by providing clarity for users.

The design of the embedded DSL controls which part of the host language users must understand. An eDSL designer has some control over which parts of the host language users will have to understand in order to use the eDSL. The decision to add a new host language feature to the list should not be taken lightly, as it has direct consequences for the user experience of the eDSL and can create barriers for users. Relatedly, designers should consider what kind of documentation and support the host language provides for a given host language feature, and how much additional instruction the eDSL designer will need to provide, before making implementation choices that expose users to additional host language functionality.

5.1.2 Users may wish to engage primarily with eDSL-specific communities, and these communities may drive adoption. Existing research indicates "developer preferences...are shaped by factors extrinsic to the language" [Meyerovich and Rabkin 2013]. While eDSL design guidelines emphasize decisions about intrinsic features of an eDSL, designers should expect significant effects on adoption and user experience if they pay attention to designing systems that support collaboration, community, and rapid or personalized assistance to users of the eDSL, as opposed to depending on host language resources alone.

5.1.3 Users may rely on existing code and avoid starting from scratch; this (i) can be supported by some eDSL features and (ii) may freeze and propagate program structures and practices. Designers should be aware that "examples will be the archetypes for thousands of programs" [Bloch 2006], and thus expect that users will treat tutorials as templates. Just as users seek out an eDSL community for collaboration and assistance, they depend on starter scripts and templates from eDSL documentation and tutorials. This may cause language practices to propagate across projects and over time. Although an eDSL designer may be tempted to lean on documentation from the host language, our participants still wanted examples of domain-specific usage. We posit that adding explicit templates to documentation (rather than only illustrative examples) may be a promising strategy for making eDSLs more learnable or usable. Designing *tutorials* with the understanding that included programs will be used as templates may also support user experience. Furthermore, eDSL designers may wish to provide (i) abstractions that support users in building upon each other's work, (ii) example creation tools (e.g. [Head et al. 2018]), or (iii) other resources that explicitly support or guide this work practice (e.g. [Ginosar et al. 2013; Kojouharov et al. 2004]).

5.1.4 Participants were sensitive to the time cost and error-proneness of data transfer between different languages and file formats. Embedded DSLs intended to perform domain-specific tasks involving data can help users avoid tedious and error-prone work by reducing or simplifying data I/O.

5.2 Discussion

A wider role and set of responsibilities for eDSL designers. Our observations suggest that when eDSL designers narrowly focus on intrinsic language features and over-rely on host language resources, eDSL user experience suffers. Instead, the designers' role may extend to encompass programming workflows, environments, tooling, information sharing, and more, which can make their role similar to a GPL designer's role. The shape of a given designer's role should be informed and shaped by the target user community's needs, preferences, priorities, and community patterns.

The climate change research community is often placed at the forefront of contentious debates with key policy implications, leading to prioritization of replicable workflows and vetted, trustworthy templates and boilerplate code [Bush et al. 2021; Feulner et al. 2016]. We observe participants looking within their own community of researchers for domain-specific templates and examples, including those provided by the Mimi designers, but rarely looking to the wider Julia community and documentation for assistance (Section 4.3). Wary of data processing mistakes, the participants benefited from the design decision to add custom data I/O functionality that kept many data flows in memory and made data easy to track (Section 4.5). Participants tended to collaborate and build on existing work, reflecting collaborative scientific practices and a tight-knit domain community. This made Mimi's modular representations useful and created a demand for workflows that support collaboration (Section 4.4.1). However, the decision to make Mimi models into packages added a level of difficulty for users trying to develop and extend models, something the designers may have been able to alleviate had they predicted the tendency to incrementally extend existing programs as opposed to simply using them out-of-the-box (Section 4.1.3).

Potential pitfalls of revealing the complexities of the host language. While the literature emphasizes the value of embedding a DSL such that users can access all features of the host GPL, and discourages making obvious distinctions between eDSL and GPL, we observed pitfalls of this approach.

Asking users of a small eDSL to engage with an array of features from the host language may degrade user experience, especially for novice programmers. Designers may benefit from considering which parts of the GPL users are *required* to understand. In fact, contrary to some of the published guidelines [Freeman and Pryce 2006; van Deursen et al. 2000], designers may want to consider *shielding* users from complex parts of the host language that could damage user experience. For the subset of host language features that are exposed to users, designers may wish to follow requests like that of P6 in Section 4.2.3 to pair documentation of the eDSL with customized documentation covering the slice of host language features that the user will need.

Overall, powerful Julia features supported both Mimi's designers and the subset of Mimi users who were most comfortable with programming, while largely hindering less comfortable programmers. Powerful Julia features—e.g., high-performance numerical computing, metaprogramming—made Mimi faster and easier to implement. That said, when these implementation details caused abstractions to leak in error messages, users faced unfamiliar vocabulary that assumed a deep understanding of the host language (Section 4.1.2) [Freeman and Pryce 2006; Kamin 1998; Kosar et al. 2008; Mernik et al. 2005]. Likewise, while using Julia packages to encapsulate Mimi Models may be useful for domain experts seeking to use a completed model, introducing Julia package development workflows to users trying to develop new models, or significantly modify existing ones, arguably damaged user experience (Section 4.1.3).

In designing eDSLs for users from non-technical domains, designers may wish to consider whether participants *want* access to the full power of a GPL. P2 emphasized that "I am not a programmer ... I'm sure I'm not very skilled but I just get things to work as I need in the moment" (P2). They did not exhibit a desire or need for complex host-language features. Novice programmers may prefer an eDSL design that actually *restricts* them—that helps keep them within narrower bounds

than the full host language. However, more advanced programmers like P5 who wish to extend the eDSL and leverage the host language features (Section 4.1.1) should have that option. This suggests eDSL designers must understand their users' backgrounds, experience, and goals in order to make informed decisions about how much to follow the conventional design guidelines of (i) allowing full access to the host language and (ii) blurring the line between eDSL and host.

Common Design Tenets For Which We Did Not Find Support. The eDSL literature commonly recommends a number of design patterns (see Section 7), and while we do find support for many of them in this work, some were not supported by our observations.

For example, existing design guidance suggests blurring the distinction between host language constructs and eDSL constructs, and intentionally matching syntax to the host language as closely as possible. Sections 4.2.1, 4.2.2, and 4.1.2 detail how blurring the distinction between host language and eDSL constructs obstructed novice Julia users [Freeman and Pryce 2006; Karsai et al. 2014; van Deursen et al. 2000]. We hypothesize that this particular design tenet might help *experienced* users of the host language, especially if they can apply their existing knowledge to make good guesses about the eDSL; however, we observed that for users with less host language experience, hiding the dividing line made debugging much harder [Denny et al. 2021; Marceau et al. 2011].

Similarly, guides encourage designers to give eDSL users access to the full power of the host language [Freeman and Pryce 2006; Kamin 1998; van Deursen et al. 2000]. Our findings do not fully contradict this design tenet, but they do temper it with evidence of the user experience consequences associated with revealing host language complexities (Section 4.2.3 and the discussion above).

Overall, we believe it may be time to reexamine and revise conventional eDSL design guidance. Although a long literature on eDSL design has (i) suggested ways to make DSL implementation easier and (ii) hypothesized about how to support DSL users, our work offers preliminary evidence that some existing guidelines may unintentionally reduce eDSL usability.

Extension of Findings to Standalone DSLs. Many of our observations may offer guidance for designers of standalone DSLs as well as embedded DSLs. Although we have attempted to focus our analysis on the elements of the user experience that are shaped not by the DSL design in isolation but specifically by the interaction of the eDSL and its host, we would not be surprised to learn that some of these lessons are applicable more broadly. For example, although we focused our discussion of error messages on cases where unfamiliar vocabulary appeared because of falling back on host language error messages, a standalone DSL can also produce unfamiliar vocabulary in its error messages. Because our contextual inquiry included only eDSL use, we do not have the data to make claims about which of our observations will generalize or how broadly. However, we hope future research may answer these questions.

6 LIMITATIONS

Role of Contextual Inquiry. Because conducting contextual inquiries is highly resource-intensive, contextual inquiry typically involves fewer participants than, for example, a controlled experiment seeking statistical significance. Guides suggest 1–20 participants [Soegaard and Dam 2012] as a general rule (See [Sauro and Lewis 2016] for a much deeper discussion of appropriate sample sizes for qualitative formative research.)

Because contextual inquiry produces such rich qualitative data, studies of this size produce data useful for suggesting important needs, gaps, and challenges. This makes contextual inquiry a good fit for research questions, like ours, that center on identifying important problems and needs. However, this approach to data collection offers no explicit insights about how to *solve* the problems it uncovers. The role of our contextual inquiry in the creation of eDSL design guidance is therefore clearly bounded. We can point designers to possible failure modes and help them anticipate user

experience pitfalls. Unfortunately, we cannot offer empirically-driven advice on how to avoid them. This work is solely focused on *need finding*.

Although our work is limited to identifying needs, we hope it can serve as a jumping off point for future research directions that go beyond need finding. Our five themes point to five spaces in which future research could test a variety of interventions to develop new, empirically justified guidance for the design of usable eDSLs.

External Validity. Because contextual inquiry centers observing participants as they complete their own work, and because our sample represents a relatively large portion of active Mimi users, we are confident that our findings are *ecologically* valid for the population of existing Mimi users. However, although Mimi users and our participants (see Table 1) come from a great variety of intellectual backgrounds, this population all has in common that they found and use Mimi. Attempting to generalize our findings to users of other eDSLs or to users who might want to *start* using eDSLs is premature. Instead, this study serves as an existence proof that our five categories of challenges and needs *can* arise in an eDSL community.

If the eDSL research community is interested in the question of whether these same challenges appear in particular eDSL communities, or whether they are common across communities, future research should study these phenomena in other eDSL user audiences. We would need to see this style of work before we could conclude that our findings are *externally* valid. Because our analysis has already identified several key areas for future research, follow-on work may be able to test the external validity with much lighter-weight data collection methods.

7 RELATED WORKS

Here we describe related works, organized into a few key categories: works that have proposed guidance about DSL user experience, but without the empirical basis of user studies; works that have studied DSL user experience via user studies; empirical studies of language adoption, not limited to DSLs; and works that identify programming needs in the climate sciences space.

7.1 DSL User Experience Literature *Without* User Studies

Our work is shaped by the long history of literature on the design of DSLs, and specifically eDSLs.

7.1.1 DSL Design Guidelines. We start with a summary of the rich work on DSL design guidelines. Many guidelines combine recommendations for making implementation easier for designers with recommendations for improving user experience for end users. We summarize common themes.

Ease of Implementation. Embedding creates powerful languages containing the features of the underlying GPL, but the DSL designer only has to put in the time and energy to design a small portion (the new abstractions or new library) [Hudak 1998; Kamin 1998; Mernik et al. 2005]. Kamin goes as far as to say "the beauty of language design by embedding is that the programming features come automatically and for free. This, in our view, is the real point of the method" [Kamin 1998].

User Access to Host Language. Choosing an embedded design adds domain-specificity while retaining the full expressive power of the host GPL [van Deursen et al. 2000]. This can be an advantage not only for designers (Section 7.1.1), but also for users who gain access to a rich GPL. In their summary of lessons learned from evolving a DSL in Java, Freeman and Pryce conclude that designers should not limit users to using the eDSL constructs [Freeman and Pryce 2006].

Constraints on Designers. Adopting a given host language means that the DSL syntax and constructs may be constrained [Cavé et al. 2010; Freeman and Pryce 2006; Mernik et al. 2005; Poltronieri et al. 2018b; van Deursen et al. 2000]. Designers of embedded DSLs may find that "the conventions

of the host language are unlikely to apply to an EDSL ... it may need to break conventions such as capitalisation, formatting, and naming for classes and methods" [Freeman and Pryce 2006]. Alternatively, choosing to write a completely new, non-embedded one would mean "no concessions are necessary regarding notation, primitives and the like" [van Deursen et al. 2000].

Error Messages. Error messages produced by eDSLs can confuse users by reflecting the concepts and vocabulary of the host language as opposed to that of the DSL [Freeman and Pryce 2006; Kamin 1998; Mernik et al. 2005]. In the worst cases, error messages can be "utterly incomprehensible [and] can be understood only by a user who not only knows [the host language], but also knows how values in the embedded language are represented" [Kamin 1998].

Overloading. Overloading can be a key tool in the creation of an embedded DSL, but can also cause confusion for users especially when the context behind a term or construct in the DSL diverges from that of the host language [Freeman and Pryce 2006; Mernik et al. 2005]. In these cases, dispatching can produce confusing error messages as covered in Section 7.1.1.

Tooling. Embedding allows a DSL to *piggyback* on existing infrastructure [Kamin 1998; Mernik et al. 2005]. However, existing research acknowledges that in many cases "editors, compilers, and debuggers are either unaware of the extensions, or must be adapted at a non-trivial cost" [Renggli et al. 2010]. Customizing IDE and other tool support for eDSLs requires engineering effort but can alleviate some usability issues [Nosál' et al. 2017].

7.1.2 Approaches for Evaluating DSL Usability. A number of works have proposed frameworks or strategies for evaluating DSL usability or user experience. In particular [Poltronieri et al. 2018a,b] propose a framework based on a focus group in which seven participants refined a framework, but without studies of DSL users. An earlier, related project suggested patterns for language designers to use for estimating their DSLs' usability [Barišić et al. 2012]. Another related project proposed a strategy based around Requirements Engineering for developing DSLs [Barišić 2017]. This work suggests strategies for engaging with users during language design work rather than lessons about language designs themselves. [Mosqueira-Rey and Alonso-Ríos 2020] proposes a set of usability heuristics specialized for DSLs. They based their heuristics on heuristics previously proposed in [Alonso-Ríos et al. 2018], but refined them to apply specifically for domain-specific contexts. No studies of DSL users played a role in the development of the heuristics.

7.1.3 Comparisons of GPLs, DSLs Without User Studies. Some works compare the usability of specific GPLs and DSLs without the use of a user study. As an example, [Cavé et al. 2010] compares the usability of the `java.util.concurrent` Java library for task parallelism with a language that natively supports task parallelism. Although these works are important food for thought, we will not cover them in depth, since they do not center user experience.

7.2 DSL User Experience Literature *With* User Studies

Studies of user experience are rare in the DSL literature. In fact, even though studies of usability are much more common than studies of user experience, even studies of *usability* are rare in the DSL literature. One systematic literature review found that among DSL construction papers at top venues, *only about 20% of papers reported usability evaluations* [Gabriel et al. 2011].

Since we are most interested in studies that specifically study *embedded* DSLS and *domain expert non-programmers*, we divide the literature according to whether: (i) the DSL under study is embedded or non-embedded, and (ii) the population under study is programmers, non-programmers, or specifically domain experts.

7.2.1 Embedded DSLs. We start with a review of user studies of embedded DSLs; we divide the literature according to whether studies include programmers, domain experts, or non-programmers more broadly.

Embedded DSLs + Programmers. To date, there have been few user studies examining DSL embedding and implications for user experience. For this discussion, we are interested in works that specifically interrogate the effects of embedding and host languages on user experience. Thus, most eDSL usability studies are out of scope; e.g., many usability studies assess task completion time with and without an eDSL. Although this is helpful for evaluating the eDSL under test, it offers little guidance for eDSL designers about the relationship between embedding and user experience.

A few studies in this space do interrogate the relationship between embedding and user experience of programmers. An analysis of Stack Overflow comments on seven Crypto libraries [Patnaik et al. 2019] characterized 16 usability issues, including some that related to embedding, and mapped them back to 10 usability principles from [Green and Smith 2016]. Another study compared ten DSL implementation approaches, including embedded, examining both designer effort and end-user effort [Kosar et al. 2008]. The authors examined end-user time, effort, and experience with debugging and error reporting.

Embedded DSLs + Domain Experts. We are not aware of any user studies of domain experts using eDSLs.

Embedded DSLs + Non-Programmers. We are aware of one study on non-programmers' user experience with eDSLs. The study in question examined two eDSL-IDE pairs, the first in Ruby with a standard IDE and the second in Java with an IDE lightly adapted to reduce common usability issues associated with eDSLs [Nosál' et al. 2017]. The authors argue that eDSLs can be usable for non-programmers, especially if designers consider both the eDSL *and* IDE support.

7.2.2 Non-Embedded DSLs. We now shift our focus to non-embedded DSLs, reviewing user studies broken down according to whether they study programmers, specifically domain experts, or other non-programmers.

Non-Embedded DSLs + Programmers. A number of works contribute user studies of non-embedded DSLs, both textual and non-textual [Albuquerque et al. 2015; Barnaby et al. 2017; Grundy et al. 2004; Ingibergsson et al. 2018; Kieburtz et al. 1996; Rao et al. 2018]. Most examine narrowly defined usability—how efficiently users can accomplish particular tasks. The works most relevant to our goals of understanding broader user experience factors are the works that study DSL readability relative to GPL readability [Cuenca et al. 2015; Ingibergsson et al. 2018; Kieburtz et al. 1996; Kosar et al. 2012].

Non-Embedded DSLs + Domain Experts. The studies described above focused on the experiences of *programmers* using DSLs. For our purposes, we are especially interested in work that centers domain experts rather than programmers. Work in this area is sparse. One work describes a (non-embedded) DSL for computer music and includes a study in which three musicians participated in a DSL evaluation process in which they read but did not construct sample programs in two DSLs [Nishino 2012]. Another studied the usability of a DSL for marine ecosystem simulation, for domain experts with a range of different educational backgrounds, academic disciplines, and programming experience levels [Johanson and Hasselbring 2017]. Another studied physicists in the domain of High Energy Physics and examined DSL readability relative to GPL readability [Barišić et al. 2011]. Finally, another studied the usability of three visual mapping languages, and participants included domain experts with a range of programming experience levels [Grundy et al. 2004]. All four of these studies used non-embedded DSLs.

Non-Embedded DSLs + Non-Programmers. In addition to work studying non-programmers with specific domain expertise, some works study non-programmers more broadly. Works that studied DSL usability for non-programmers—but *not* domain experts—include [Aleven et al. 2016; Alexandrova et al. 2015; Barišić et al. 2018; Elsts et al. 2013; Parham-Mocello et al. 2022; Rodríguez-Gil et al. 2019; Taipalus and Grahn 2023]. Some of these works conducted user studies to assess the usability of *non-textual* DSLs for domain experts and non-programmers—e.g. [Aleven et al. 2016; Alexandrova et al. 2015; Rodríguez-Gil et al. 2019]. Even including visual DSLs, there are still relatively few DSL user studies of non-programmers. None of the visual DSLs studied via user studies are eDSLs.

7.3 User Studies of Adoption

A small set of works have assessed characteristics of programming languages that affect adoption. Developers rate evidence of reliable and active communication channels as important factors when choosing open-source packages [Head 2016]. A survey-based study found that for general-purpose languages "social factors outweigh intrinsics" [Meyerovich and Rabkin 2013] and identified features like usefulness or richness of available libraries, and existing code bases as significant for adoption.

7.4 Identifying Programming Needs within Climate Sciences

Software is increasingly integral to cutting-edge research across the earth sciences, and particularly the climate change domain [Balaji et al. 2018; Bush et al. 2021; Feulner et al. 2016; Gentemann et al. 2021; Williams 2014]. The growing diversity of researchers and policy makers in the climate change space increases the demand for domain-specific tools, and some researchers have responded by mapping out areas for contribution by the computer science community [Easterbrook 2010].

The case for programming tool contributions to this domain is gaining traction, but the paucity of user studies examining user needs creates a barrier to impact. Easterbrook's 2009 ethnographic study of climate scientists examines one team's software development culture and practices—of the related works, this is the work that comes closest to touching on the topic of programmer needs and user experience in the climate sciences [Easterbrook and Johns 2009].

8 CONCLUSIONS AND FUTURE WORK

The embedded approach to DSL implementation—developing a DSL within a general-purpose host language—allows designers to quickly develop powerful languages for niche domains and serve the needs of a diverse, growing body of experts. Our work demonstrates that many eDSL design decisions are relevant not only for the eDSL developer, but also the eventual user experience. As we develop a better understanding of eDSL user experience, we can direct future research and design work towards addressing usability barriers. Our contextual inquiry with domain experts using a particular eDSL uncovered five promising directions in particular. For each of these directions, future work could, for example: (i) explore how these patterns vary across choice of host language, choice of embedded language, and choice of target audience or (ii) contribute controlled experiments or other studies to rigorously test the patterns we observed. We are still in the relatively early stages of understanding the effects of language implementation choices on non-traditional programming audiences. We hope future studies will expand and deepen our understanding of how eDSLs can meet the programming needs of the large and varied body of domain experts.

ACKNOWLEDGMENTS

We would like to thank the anonymous participants for their time and effort. We would also like to thank the following people for their generous support, advice, and contributions: Prof. David Anthoff, the PLAIT lab at UC Berkeley.

REFERENCES

Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahão, and António Ribeiro. 2015. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software* 101 (2015), 245–259. https://doi.org/10.1016/j.jss.2014.11.051

Vincent Aleven, Ryan Baker, Yuan Wang, Jonathan Sewall, and Octav Popescu. 2016. Bringing non-programmer authoring of intelligent tutors to MOOCs. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. 313–316. https://doi.org/10.1145/2876034.2893442

Sonya Alexandrova, Zachary Tatlock, and Maya Cakmak. 2015. RoboFlow: A flow-based visual programming language for mobile manipulation tasks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 5537–5544. https://doi.org/10.1109/ICRA.2015.7139973

David Alonso-Ríos, Eduardo Mosqueira-Rey, and Vicente Moret-Bonillo. 2018. A systematic and generalizable approach to the heuristic evaluation of user interfaces. *International Journal of Human–Computer Interaction* 34, 12 (2018), 1169–1182. https://doi.org/10.1080/10447318.2018.1424101

David Anthoff. 2020. *JuliaCon 2020 | Using VS Code for Julia Development| David Anthoff - YouTube*. https://www.youtube.com/watch?v=IdhnP00Y1Ks Accessed on 2022-07-16.

David Anthoff. 2022. *Home - VegaLite.jl*. https://www.queryverse.org/VegaLite.jl/stable/ Accessed on 2022-8-18.

David Anthoff, Lisa Rennels, Cora Kingdon, and Richard Plevin. 2019. *Mimi Framework*. https://forum.mimiframework.org Accessed on 2022-07-16.

David Anthoff, Lisa Rennels, Cora Kingdon, and Richard Plevin. 2023a. *Github - mimiframework/Mimi.jl: Integrated Assessment Modeling Framework*. https://github.com/mimiframework Accessed on 2022-07-16.

David Anthoff, Lisa Rennels, Cora Kingdon, and Richard Plevin. 2023b. *Github - mimiframework/Mimi.jl: Integrated Assessment Modeling Framework - Models as Packages*. https://www.mimiframework.org/Mimi.jl/stable/explanations/exp_pkgs Accessed on 2022-08-07.

David Anthoff, Lisa Rennels, Cora Kingdon, and Richard Plevin. 2023c. *Github - mimiframework/Mimi.jl: Integrated Assessment Modeling Framework - Models as Packages*. https://www.mimiframework.org/Mimi.jl/stable/tutorials/tutorial_2/# Accessed on 2022-08-07.

Venkatramani Balaji, Karl E Taylor, Martin Juckes, Bryan N Lawrence, Paul J Durack, Michael Lautenschlager, Chris Blanton, Luca Cinquini, Sébastien Denvil, Mark Elkington, et al. 2018. Requirements for a global data infrastructure in support of CMIP6. *Geoscientific Model Development* 11, 9 (2018), 3659–3680. https://doi.org/10.5194/gmd-11-3659-2018

Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for domain-specific languages. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29. https://doi.org/10.1145/3428297

Ankica Barišić, Vasco Amaral, Miguel Goulao, and Bruno Barroca. 2011. Quality in use of domain-specific languages: a case study. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*. 65–72. https://doi.org/10.1145/2089155.2089170

Ankica Barišić, João Cambeiro, Vasco Amaral, Miguel Goulão, and Tarquínio Mota. 2018. Leveraging teenagers feedback in the development of a domain-specific language: the case of programming low-cost robots. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1221–1229. https://doi.org/10.1145/3167132.3167264

Ankica Barišić. 2017. Framework Support for Usability Evaluation of Domain-Specific Languages. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Vancouver, BC, Canada) *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 16–18. https://doi.org/10.1145/3135932.3135953

Ankica Barišić, Pedro Monteiro, Vasco Amaral, Miguel Goulão, and Miguel Monteiro. 2012. Patterns for Evaluating Usability of Domain-Specific Languages. In *Proceedings of the 19th Conference on Pattern Languages of Programs* (Tucson, Arizona) *(PLoP '12)*. The Hillside Group, USA, Article 14, 34 pages. https://doi.org/10.5555/2821679.2831284

Celeste Barnaby, Michael Coblenz, Tyler Etzel, Eliezer Kanal, Joshua Sunshine, Brad Myers, and Jonathan Aldrich. 2017. A user study to inform the design of the obsidian blockchain DSL. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'17)*.

Bernardo A Bastien-Olvera and Frances C Moore. 2021. Use and non-use value of nature and the social cost of carbon. *Nature Sustainability* 4, 2 (2021), 101–108. https://doi.org/10.1038/s41893-020-00615-0

Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education* (2019), 177–210. https://doi.org/10.1145/3344429.3372508

Jeff Bezanson. 2019. *JuliaCon2019 | What's Bad About Julia | Jeff Bezanson - YouTube*. https://www.youtube.com/watch?v=TPuJsgyu87U Accessed on 2022-09-09.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

Alan Blackwell and Thomas Green. 2003. Notational systems–the cognitive dimensions of notations framework. *HCI models, theories, and frameworks: toward an interdisciplinary science. Morgan Kaufmann* (2003). https://doi.org/10.1016/B978-155860808-5/50005-8

Joshua Bloch. 2006. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 506–507. https://doi.org/10.1145/1176617.1176622

Emma Boudreau. 2019. *The Serious Downsides To The Julia Language In 1.0.3 | by Emma Boudreau | Towards Data Science.* https://towardsdatascience.com/the-serious-downsides-to-the-julia-language-in-1-0-3-e295bc4b4755 Accessed on 2022-07-16.

Emma Boudreau. 2021. *The Depressing Challenges Facing The Julia Programming Language In 2021 | by Emma Boudreau | Towards Data Science.* https://towardsdatascience.com/the-depressing-challenges-facing-the-julia-programming-language-in-2021-34c748968ab7 Accessed on 2022-08-31.

Claus Brabrand and Michael I Schwartzbach. 2002. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. 31–40. https://doi.org/10.1145/503032.503035

Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

Rosemary Bush, Andrea Dutton, Michael Evans, Rich Loft, and Gavin A Schmidt. 2021. Perspectives on Data Reproducibility and Replicability in Paleoclimate and Climate Science. *Harvard Data Science Review* 2, 4 (2021). https://doi.org/10.1162/99608f92.00cd8f85

Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2010. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*. 1–6. https://doi.org/10.1145/1937117.1937126

Sarah Chasins. 2019. *Democratizing Web Automation: Programming for Social Scientists and Other Domain Experts*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-139.html

Wikipedia Conributors. 2022. *Multiple Dispatch - Wikipedia.* https://en.wikipedia.org/wiki/Multiple_dispatch Accessed on 2022-08-31.

Andrew Crabtree, Mark Rouncefield, and Peter Tolmie. 2012. *Doing design ethnography*. Springer. https://doi.org/10.1007/978-1-4471-2726-0

Fredy Cuenca, Jan Van den Bergh, Kris Luyten, and Karin Coninx. 2015. A user study for comparing the programming efficiency of modifying executable multimodal interaction descriptions: a domain-specific language versus equivalent event-callback code. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. 31–38. https://doi.org/10.1145/2846680.2846686

Ryan Culpepper. 2012. Fortifying macros. *Journal of functional programming* 22, 4-5 (2012), 439–476. https://doi.org/10.1017/S0956796812000275

Ryan Culpepper and Matthias Felleisen. 2004. Taming macros. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings 3*. Springer, 225–243. https://doi.org/10.1007/978-3-540-30175-2_12

Ryan Culpepper and Matthias Felleisen. 2007. Debugging macros. In *Proceedings of the 6th international conference on Generative programming and component engineering*. 135–144. https://doi.org/10.1016/j.scico.2009.06.001

Jaan Tollander de Balsch. 2021. *How to Create Software Packages with Julia Language | Jaan Tollander de Balsch.* https://jaantollander.com/post/how-to-create-software-packages-with-julia-language/ Accessed on 2022-07-16.

Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 55, 15 pages. https://doi.org/10.1145/3411764.3445696

Gergely Dévai, Dániel Leskó, and Máté Tejfel. 2015. The EDSL's struggle for their sources. *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers 5* (2015), 300–335. https://doi.org/10.1007/978-3-319-15940-9_7

Simon Dietz, James Rising, Thomas Stoerk, and Gernot Wagner. 2021. Economic impacts of tipping points in the climate system. *Proceedings of the National Academy of Sciences* 118, 34 (2021), e2103081118. https://doi.org/10.1073/pnas.2103081118

DSB. 2020. *Developing your Julia package. A Tutorial on how to quickly and easily develop your own Julia package. | by DSB | Coffee in a Bottle | Medium.* https://medium.com/coffee-in-a-klein-bottle/developing-your-julia-package-682c1d309507 Accessed on 2022-07-16.

Steve M Easterbrook. 2010. Climate change: a grand software challenge. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 99–104. https://doi.org/10.1145/1882362.1882383

Steve M Easterbrook and Timothy C Johns. 2009. Engineering the software for understanding climate change. *Computing in science & engineering* 11, 6 (2009), 65–74. https://doi.org/10.1109/MCSE.2009.193

Atis Elsts, Janis Judvaitis, and Leo Selavo. 2013. SEAL: a domain-specific language for novice wireless sensor network programmers. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 220–227. https://doi.org/10.1145/3607180

Frank C Errickson, Klaus Keller, William D Collins, Vivek Srikrishnan, and David Anthoff. 2021. Equity is more important for the social cost of methane than climate uncertainty. *Nature* 592, 7855 (2021), 564–570. https://doi.org/10.1038/s41586-021-03386-6

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press. https://doi.org/10.5555/3265452

Georg Feulner, H Atmanspacher, and S Maasen. 2016. Science under Societal Scrutiny: Reproducibility in Climate Science. In *Reproducibility: Principles, Problems, Practices, and Prospects*. Wiley, 269–285. https://doi.org/10.1002/9781118865064.ch12

Steve Freeman and Nat Pryce. 2006. Evolving an embedded domain-specific language in Java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 855–865. https://doi.org/10.1145/1176617.1176735

Pedro Gabriel, Miguel Goulao, and Vasco Amaral. 2011. Do software languages engineers evaluate their languages? *arXiv preprint arXiv:1109.6794* (2011). https://doi.org/10.48550/arXiv.1109.6794

Chelle Leigh Gentemann, Chris Holdgraf, Ryan Abernathey, Daniel Crichton, James Colliander, Edward Joseph Kearns, Yuvi Panda, and Richard P Signell. 2021. Science storms the cloud. *AGU Advances* 2, 2 (2021), e2020AV000354. https://doi.org/10.1029/2020AV000354

Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Bjorn Hartmann. 2013. Authoring multi-stage code examples with editable code histories. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 485–494. https://doi.org/10.1145/2501988.2502053

Charles Goodwin. 2015. Professional vision. In *Aufmerksamkeit: Geschichte-Theorie-Empirie*. Springer, 387–425. https://doi.org/10.1007/978-3-531-19381-6_20

Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. 2008. DSLs: the good, the bad, and the ugly. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. 791–794. https://doi.org/10.1145/1449814.1449863

Matthew Green and Matthew Smith. 2016. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy* 14, 5 (2016), 40–46. https://doi.org/10.1109/MSP.2016.111

John C Grundy, John G Hosking, RW Amor, Warwick B Mugridge, and Yongqiang Li. 2004. Domain-specific visual languages for specifying and generating data mapping systems. *Journal of Visual Languages & Computing* 15, 3-4 (2004), 243–263. https://doi.org/10.1016/j.jvlc.2004.01.003

Ian Hacking. 1991. A tradition of natural kinds. *Philosophical Studies: An International Journal for Philosophy in the Analytic Tradition* 61, 1/2 (1991), 109–126.

Andrew Head. 2016. Social health cues developers use when choosing open source packages. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 1133–1135. https://doi.org/10.1145/2950290.2983973

Andrew Head, Elena L Glassman, Björn Hartmann, and Marti A Hearst. 2018. Interactive extraction of examples from existing code. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12. https://doi.org/10.1145/3173574.3173659

Zef Hemel, Danny M Groenewegen, Lennart CL Kats, and Eelco Visser. 2011. Static consistency checking of web applications with WebDSL. *Journal of Symbolic Computation* 46, 2 (2011), 150–182. https://doi.org/10.1016/j.jsc.2010.08.006

Karen Holtzblatt and Hugh Beyer. 1997. *Contextual design: defining customer-centered systems*. Elsevier.

K Holzblatt and Hugh Beyer. 2017. Contextual design: Design for Life. https://doi.org/10.1007/978-3-031-02207-4_2

Paul Hudak. 1998. Modular domain specific languages and tools. In *Proceedings. Fifth international conference on software reuse (Cat. No. 98TB100203)*. IEEE, 134–142. https://doi.org/10.1109/ICSR.1998.685738

Johann Thor Mogensen Ingibergsson, Stefan Hanenberg, Joshua Sunshine, and Ulrik Pagh Schultz. 2018. Experience report: Studying the readability of a domain specific language. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2030–2033. https://doi.org/10.1145/3167132.3167436

Arne N Johanson and Wilhelm Hasselbring. 2017. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering* 22 (2017), 2206–2236. https://doi.org//10.1007/s10664-016-9483-z

Julia Contributors. 2022a. *3. Managing Packages - Pkg.jl*. https://pkgdocs.julialang.org/v1/managing-packages/#developing Accessed on 2022-08-31.

Julia Contributors. 2022b. *Methods - The Julia Language*. https://docs.julialang.org/en/v1/manual/methods/ Accessed on 2022-08-31.

Juno. 2022. *Juno.* https://junolab.org Accessed on 2022-8-18.

Project Jupyter. 2022. *Project Jupyter | Home.* https://jupyter.org Accessed on 2022-8-18.

Samuel N Kamin. 1998. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science* 14 (1998), 149–168. https://doi.org/10.1016/S1571-0661(05)80235-X

Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2014. Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378* (2014). https://doi.org/10.48550/arXiv.1409.2378

Richard B Kieburtz, Laura McKinney, Jeffrey M Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P Oliva, Tim Sheard, Ira Smith, and Lisa Walton. 1996. A software engineering experiment in software component generation. In *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE, 542–552. https://doi.org/10.1109/ICSE.1996.493448

Eleanor Knott, Aliya Hamid Rao, Kate Summers, and Chana Teeger. 2022. Interviews in the social sciences. *Nature Reviews Methods Primers* 2, 1 (2022), 73. https://doi.org/10.1038/s43586-022-00150-6

Chris Kojouharov, Aleksey Solodovnik, and Gleb Naumovich. 2004. Jtutor: an eclipse plug-in suite for creation and replay of code-based tutorials. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. 27–31. https://doi.org//10.1145/1066129.1066135

Tomaž Kosar, Pablo E Martı, Pablo A Barrientos, Marjan Mernik, et al. 2008. A preliminary study on various implementation approaches of domain-specific language. *Information and software technology* 50, 5 (2008), 390–405. https://doi.org/10.1016/j.infsof.2007.04.002

Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering* 17, 3 (2012), 276–304. https://doi.org/10.1007/s10664-011-9172-x

Shriram Krishnamurthi and Tim Nelson. 2019. The human in formal methods. In *International Symposium on Formal Methods*. Springer, 3–10. https://doi.org/10.1007/978-3-030-30942-8_1

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 499–504. https://doi.org/10.1145/1953163.1953308

Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892

Leo A Meyerovich and Ariel S Rabkin. 2013. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 1–18. https://doi.org/10.1145/2509136.2509515

Microsoft. 2022. *Visual Studio Code. Code Editing. Redefined.* https://code.visualstudio.com Accessed on 2022-8-18.

Eduardo Mosqueira-Rey and David Alonso-Ríos. 2020. Usability Heuristics for Domain-Specific Languages (DSLs). In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1340–1343. https://doi.org/10.1145/3341105.3374234

NASEM. 2017. *Valuing climate damages: updating estimation of the social cost of carbon dioxide.* National Academies Press. https://doi.org/10.17226/24651

Eric Niebler. 2007. Proto: A compiler construction toolkit for DSELs. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*. 42–51. https://doi.org/10.1145/1512762.1512767

Hiroki Nishino. 2012. How can a DSL for expert end-users be designed for better usability? a case study in computer music. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*. 2673–2678. https://doi.org/10.1145/2212776.2223855

Jakonb Nybo Nissen. 2022a. *How to optimise Julia code: A practical guide.* https://viralinstruction.com/posts/optimise/ Accessed on 2022-09-12.

Jakonb Nybo Nissen. 2022b. *What's bad about Julia?* https://viralinstruction.com/posts/badjulia/ Accessed on 2022-08-31.

William Nordhaus. 1982. How fast should we graze the global commons? *The American Economic Review* 72, 2 (1982), 242–246.

Milan Nosál', Jaroslav Porubän, and Matúš Sulír. 2017. Customizing host IDE for non-programming users of pure embedded DSLs: A case study. *Computer Languages, Systems & Structures* 49 (2017), 101–118. https://doi.org/10.1016/j.cl.2017.04.003

Jennifer Parham-Mocello, Aiden Nelson, and Martin Erwig. 2022. Exploring the Use of Games and a Domain-Specific Teaching Language in CS0. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*. 351–357. https://doi.org/10.1145/3502718.3524812

Nikhil Patnaik, Joseph Hallett, and Awais Rashid. 2019. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. 245–257. https://www.usenix.org/conference/soups2019/presentation/patnaik

Ildevana Poltronieri, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. 2018a. Usa-DSL: Usability Evaluation Framework for Domain-Specific Languages. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 2013–2021. https:

//doi.org/10.1145/3167132.3167348

Ildevana Poltronieri, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. 2018b. Usability evaluation framework for domain-specific language: A focus group study. *ACM SIGAPP Applied Computing Review* 18, 3 (2018), 5–18. https://doi.org/10.1145/3284971.3284973

David Randall, Richard Harper, and Mark Rouncefield. 2007. *Fieldwork for design: theory and practice.* Springer Science & Business Media. https://doi.org/10.1007/978-1-84628-768-8

Arjun Rao, Ayush Bihani, and Mydhili Nair. 2018. Milo: A visual programming environment for data science education. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, 211–215. https://doi.org/10.1109/VLHCC.2018.8506504

Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. 2010. Embedding languages without breaking tools. In *European Conference on Object-Oriented Programming.* Springer, 380–404. https://doi.org/10.1007/978-3-642-14107-2_19

Kevin Rennert, Frank Errickson, Brian C Prest, Lisa Rennels, Richard G Newell, William Pizer, Cora Kingdon, Jordan Wingenroth, Roger Cooke, Bryan Parthum, et al. 2022a. Comprehensive evidence implies a higher social cost of CO2. *Nature* 610, 7933 (2022), 687–692. https://doi.org/10.1038/s41586-022-05224-9

Kevin Rennert, Brian C Prest, William A Pizer, Richard G Newell, David Anthoff, Cora Kingdon, Lisa Rennels, Roger Cooke, Adrian E Raftery, Hana Ševčíková, et al. 2022b. The social cost of carbon: advances in long-term probabilistic projections of population, GDP, emissions, and discount rates. *Brookings Papers on Economic Activity* 2021, 2 (2022), 223–305. https://doi.org/10.1353/eca.2022.0003

Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34. https://doi.org/10.1109/MS.2009.193

Martin P Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16 (2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

Luis Rodríguez-Gil, Javier García-Zubia, Pablo Orduña, Aitor Villar-Martinez, and Diego López-De-Ipiña. 2019. New approach for conversational agent definition by non-programmers: a visual domain-specific language. *IEEE Access* 7 (2019), 5262–5276. https://doi.org/10.1109/ACCESS.2018.2883500

Marc Sacks. 1994. *On the Job Learning in the Software Industry: Corporate Culture and the Acquisition of Knowledge.* Greenwood Publishing Group Inc. https://doi.org/10.1080/08109029608632033

Jeff Sauro and J Lewis. 2016. Chapter 7: What sample sizes do we need? Part 2: Formative studies. *Quantifying the User Experience: Practical Statistics for User Research. Cambridge, MA: Morgan Kaufmann-Elsevier Science and Technology Books, Inc* 3 (2016). https://doi.org/10.1016/B978-0-12-802308-2.00007-2

Mads Soegaard and Rikke Friis Dam. 2012. The encyclopedia of human-computer interaction. *The encyclopedia of human-computer interaction* (2012).

VERBI Software. 2022. MAXQDA. https://maxqda.com/. Accessed: 2022-11-17.

Toni Taipalus and Hilkka Grahn. 2023. Framework for SQL Error Message Design: A Data-Driven Approach. *ACM Transactions on Software Engineering and Methodology* (2023). https://doi.org/10.1145/3607180

Kyle Thayer, Sarah E Chasins, and Amy J Ko. 2021. A theory of robust API knowledge. *ACM Transactions on Computing Education (TOCE)* 21, 1 (2021), 1–32. https://doi.org/10.1145/3444945

US Environmental Protection Agency. 2022a. EPA External Review Draft of Report on the Social Cost of Greenhouse Gases: Estimates Incorporating Recent Scientific Advances. (2022). https://www.epa.gov/system/files/documents/2022-11/epa_scghg_report_draft_0.pdf

US Environmental Protection Agency. 2022b. Regulatory Impact Analysis for Phasing Down Production and Consumption of Hydrofluorocarbons (HFCs). (2022). https://www.epa.gov/system/files/documents/2022-07/RIA%20for%20Phasing%20Down%20Production%20and%20Consumption%20of%20Hydrofluorocarbons%20%28HFCs%29.pdf

Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.* 35, 6 (jun 2000), 26–36. https://doi.org/10.1145/352029.352035

Guri Verne and Tone Bratteteig. 2018. Inquiry when doing research and design: Wearing two hats. *IxD&A* 38 (2018), 89–106.

Hadley Wickham. 2016. Data analysis. In *ggplot2.* Springer, 189–201. https://doi.org/10.1007/978-3-319-24277-4_9

Dean N Williams. 2014. Visualization and analysis tools for ultrascale climate data. *Eos, Transactions American Geophysical Union* 95, 42 (2014), 377–378. https://doi.org/10.1002/2014EO420002

Larry E Wood. 1997. Semi-structured interviewing for user-centered design. *Interactions* 4, 2 (1997), 48–61. https://doi.org/10.1145/245129.245134

Received 2023-04-14; accepted 2023-08-27