Contextual Inquiry into Programmers' Use of Mimi and Implications for Embedded DSL Design

Lisa Rennels
Sarah Chasins

Contextual Inquiry into Programmers' Use of Mimi and Implications for Embedded DSL Design

LISA RENNELS, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, 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 usability 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 four key themes, including: the host language has significant and sometimes unexpected impacts on eDSL usability, and users preferentially engage with domain-specific communities and code templates rather than host language resources. The needs uncovered in our study offer implications for future eDSL designs and suggest directions for future language 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, usability, contextual inquiry, need finding

1 INTRODUCTION

Experts across a diverse range of fields are coming to rely on programming and computation. We see this trend playing out across myriad domains, including earth sciences and climate change [19, 29, 30, 62]. For domains like these, in which practitioners typically lack formal computer science education, the usability of available languages is a substantial pain point. These users typically do not share the body of computing-related background we expect from professional or formally trained programmers. 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 [33, 60]. Embedded domain-specific languages (eDSLs) *embed* new constructs in a host 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 language, embedding is efficient for the designer and provides a rich, extensible tool for the end user, although it comes with limitations and complications introduced by the host language [26, 34, 43, 49]. Although there are a range of different ways to embed languages, often distinguished by terms such as shallowly or deeply embedded languages, this paper will focus on the kinds of DSLs that mostly function as libraries within their host language.

The embedded approach is popular with DSL designers, in large part because of ease of implementation, as discussed in Section 2.4. While eDSLs can be immensely valuable, usability challenges may make eDSLs less appealing to *users* than to *designers* [37]. One empirical study of ten DSL implementations (9 non-embedded DSLs and one eDSL) found that while the eDSL ranked best in *implementation effort*, it ranked second worst in *end-user effort* [46]. This points to a

Authors' addresses: Lisa Rennels, lrennels@berkeley.edu, University of California, Berkeley, Berkeley, California, USA, 94720; Sarah E. Chasins, schasins@berkeley.edu, University of California, Berkeley, Berkeley, California, USA, 94720.

missed opportunity for language designers that could be addressed with a better understanding of the needs of eDSL users.

We have only a small body of literature observing how users *actually* interact with embedded DSLs (Section 2.5). By adding to this area of research, we hope to support the community in improving eDSL usability, making eDSLs attractive not only for designers but also for users. Methods for conducting usability analyses on DSLs necessarily differ from methods for GPLs, in part because the users of DSLs are domain experts and not necessarily computer scientists or even practiced programmers. Thus, priorities and metrics for success may differ [20]. Characteristics like the relationship between an eDSL and its host language are unique to the embedded approach and demand fresh approaches for understanding language usability.

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

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

For example, we observe that error messages that reference host-language concepts unfamiliar to users, like intricacies of the type system or expression parsing, make eDSL abstractions difficult for users to debug. Similarly, the eDSL design forced many users to engage with advanced package development workflows, requiring an additional layer of host language knowledge and imposing a potentially unnecessary burden on users. Based on our findings, we suggest a set of design considerations for future eDSL designers.

2 BACKGROUND AND RELATED WORK

Here we briefly introduce the particular eDSL we studied in this work and the history of research on the usability of languages, DSLs, and eDSLs.

2.1 Mimi and Integrated Assessment Models

Integrated assessment models (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 significant utility and contributions of these modeling efforts were recently highlighted by the award of the 2018 Nobel Memorial Prize in Economic Sciences to Professor William Nordhaus, who pioneered these macroeconomic models in the early 1980s [53].

Unfortunately, the fragmented multitude of hosting platforms, programming languages, and APIs used by IAM authors makes the task of tackling analysis of one, let alone several, IAMs daunting for even experienced researchers. This setup also forces researchers and interested parties to write their own code to carry out crucial steps like uncertainty and sensitivity analysis, with the result that practitioners often skip it or perform it haphazardly. A seminal report by the National Academies of Sciences, Engineering, and Medicine entitled "Valuing Climate Damages: Updating Estimation of the Social Cost of Carbon Dioxide" recommended the development of “an open-source, computationally efficient, publicly accessible, and clearly documented computational platform” to help remedy these problems [51].

In our study, we observe users of Mimi [1], a domain-specific language embedded within the general-purpose Julia language. Mimi aims to fill the need for a uniform, integrated software platform that co-locates a variety of current and in-development IAMs. Mimi uses a custom set of abstractions to streamline model creation and editing, enables collaboration with a modular approach, and provides infrastructure for sensitivity analysis and results visualization. It leverages Julia features including Julia syntax, multiple dispatch, dynamic typing, high performance numerical computing, and advanced environment and package management support [23].

2.2 Languages and Tools for Earth Sciences and Climate Change

Software is increasingly integral to cutting-edge research in the earth sciences, and particularly to the climate change domain [19, 25, 32, 36, 62]. The growing assortment of researchers and policy makers in the climate change discipline increases the demand for domain-specific tools, and some members of the computer science community have responded by mapping out areas for contribution [29].

The case for contributions to this domain from the computing community is gaining traction, but the paucity of user studies examining user needs creates a barrier to action. Easterbrook's 2009 ethnographic study of climate scientists examines one team's software development culture and practices—this is the work that comes closest to touching on the topic of programmer needs in the climate sciences [30].

2.3 User Studies of Adoption and Feature Importance for General Purpose Languages

Intrinsic and extrinsic features of a language, such as community engagement, available expertise, and the surrounding library ecosystem are important factors for usability. Developers rate evidence of reliable and active communication channels as important factors when choosing open-source packages [40]. One key empirical study suggested "social factors outweigh intrinsics" [50] and identified features like usefulness or richness of available libraries, and existing code bases as significant for adoption. Although these works did not explicitly touch on embedded domain-specific languages, some lessons from general purpose languages may extend to eDSLs.

2.4 DSL Usability Literature *Without* User Studies

A long-standing body of literature on the design of domain-specific languages, and specifically on the embedded approach, informs our study.

Ease of Design. 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) [43, 44, 49]. 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" [44].

User Access to Host Language. Choosing an embedded design adds domain-specificity while retaining the full expressive power of the host GPL [60]. This can be an advantage not only for designers (Section 2.4), 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 [34].

Constraints on Designers. Adopting a given host language means that the DSL syntax and constructs may be constrained [26, 34, 49, 56, 60]. 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" [34]. Alternatively, choosing to write a completely new, non-embedded one would mean "no concessions are necessary regarding notation, primitives and the like" [60].

Error Messages. Error messages produced by embedded DSLs are often confusing for users because they reflect the concepts and vocabulary of the host language as opposed to that of the domain-specific language [34, 44, 49]. 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" [44].

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 [34, 49]. In these cases, it is possible than such dispatching can produce confusing error messages as mentioned in Section 2.4.

Tooling. Embedding allows a DSL to *piggyback* on existing infrastructure [44, 49]. 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" [57]. Customizing IDE and other tool support for eDSLs requires engineering effort but can alleviate some usability issues [54].

2.5 DSL Usability Literature *With* User Studies

User studies of usability are rare in the DSL literature, with one systematic literature review finding that *only about 20% of reviewed papers reported usability evaluations* [35].

2.5.1 Embedded DSLs. To date, there have been few user studies assessing eDSL design and implications for usability. One thematic analysis of Stack Overflow comments on seven different Crypto libraries outlined 16 unique issues, mapped them back to 10 usability principles (from [38]), and then grouped the issues into four different usability smells [55]. Another empirical study examined the usability of a Java-based eDSL paired with a customized NetBeans IDE, adapted to reduce common usability issues associated with embedded DSLs, compared against a Ruby-based eDSL paired with a non-customized IDE [54]. The authors assessed whether their IDE improvements outweighed the advantages of Ruby's syntactic flexibility. A somewhat older study compares ten DSL implementation approaches, including embedded, examining both designer effort and end-user effort [46]. The authors examined end-user time, effort, and experience with debugging and error reporting. We are not aware of other studies of eDSL usability.

2.5.2 Non-Embedded DSLs. The literature includes a number of studies employing empirical usability analysis of non-embedded DSLs [20, 22, 39, 42, 45, 52, 58]. For example, the creators of the Gyro Creator Language (GCL), a robotics DSL for teens, compared the usability of their language to contemporaries Lego and Scratch using video-recorded programming challenges and 5-minute feedback sessions [21]. Another work surveyed and interviewed participants about two DSLs—Detex and Tamdera—and their relative usability for software maintenance [18]. A few additional empirical studies compare the usability of DSLs versus GPLs [28, 47].

3 STUDY DESIGN

We conducted a contextual inquiry [41] with nine participants and analyzed it via thematic analysis.

Participants and Recruitment. We recruited nine participants via snowball sampling, starting with recruitment emails to individuals and groups and a recruitment post to the public Mimi user forum [2]. The nine participants represented a variety of industries, years of experience in the climate change domain, and levels of formal and informal education (Figure 1). All participants were existing Mimi users who brought their own Mimi projects for their sessions.

ID	Age	Employment	Years in Climate Change Domain	Setting for Mimi.jl Work	Programming Education (formal & informal)	PL and Tools (used in last 2 weeks)
P1	30	PhD Candidate, Geography	7	academic research	- Numerical modeling courses (BSc)	R
P2	32	Post-Doctoral Researcher, School of Government	10	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	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	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	academic research	- Formal education in C++, Scheme	R, Python, Julia, Javascript
P6	59	Associate Professor, Department of Computational Data Sciences	30	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	academic research	- Government - Statistics undergraduate degree (R) - Online courses (Python)	R, Julia, Python
P8	28	Research Associate at NGO	1	NGO	- Data science course (graduate school) - R-based ecological dynamics course	R, Julia
P9	33	Assistant Professor, School of Mathematical Sciences	11	academic research	- Introductory computer science (undergraduate) - Learning on the job and teaching Computer/Data Science	Julia, Python, R, Excel, GitHub, Google Colab, Jupyter notebooks

Table 1. Key information about study participants.

Consent and Session Protocol. Each session began with a short introduction to the study and reaffirmation of informed consent, which each participant read and signed before the session in accordance with our institutional review board. We then conducted a 50-minute contextual inquiry during which participants worked on an existing Mimi project of their choosing and were asked to speak out loud to help the researcher understand what they were doing. The session concluded with a 5-10-minute semi-structured interview.

Analysis. We performed an inductive, semantic thematic analysis of the audiovisual data, using open coding to attach short descriptive labels, or codes, to segments of the videos, then iteratively build up a hierarchy of observed themes.

4 FINDINGS

In this section we present our key findings, organized into four high-level themes.

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" (P9). While the literature emphasizes that choice of host GPL has consequences for eDSL developers [34, 44, 49], this section focuses on the implication of host language choice *on eDSL users*.

4.1.1 Host Language Package Management System. Selecting a host language also typically means selecting a package management system, since many eDSLs are implemented as packages for the host language. Since Mimi is embedded in Julia, participants had to learn and engage with the Julia package management system.

When asked about use of environments, P3 stated that they "love the Julia environments" (P3), though they took some effort to learn. In fact, "a lot of [their] questions about how to do things are *workflow* as much as anything else" (P3). P7 emphasized that learning curve:

> "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)

During their session, P4 seamlessly and frequently used the `Pkg> st` call in the REPL to check what packages, and in 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 on their project and 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 verbose process of re-adding packages to 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. 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. The visibility into environment state, verbosity, and slow precompilation time of Julia package management are common complaints in the Julia community [3], and we saw these concerned echoed by Mimi users. The design opportunities for eDSL designers are two-fold: (1) designers may consider assessing the usability specifically of candidate host languages' package management systems, especially if they plan to implement their eDSL as a package; (2) designers may wish to identify commonly-known host language usability issues in order to preemptively smooth out the experience for their eDSL users.

4.1.2 Availability and Stability of Other Tools for the Host Language. The participant experience was substantially affected by the availability and stability of Julia packages, as well as their ability to integrate into the larger ecosystem of traditional tools and infrastructure like Github and popular IDEs (see Section 2.4 for related literature).

Since Julia is a fairly young language, 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)

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 [61] to do plotting. This incurred the cost of data I/O, described in detail in Section 4.3. 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 [11] for data exploration and plotting. The participant often paused to get their bearings and created disparate files that were difficult to track, as demonstrated by the fact that P9 spent the first fifteen minutes of their session searching for pre-existing work and files.

P5, an advanced Mimi user, took advantage of Julia packages for optimization and linear programming. They said that this integration had thus far been "working quite well" (P5) for them. This was key to making Mimi useful for their work, and leveraged the strengths of the Julia host language in numerical programming and performance. Both P4 and P9 similarly leveraged Julia's rich ecosystem of statistical packages and used them to provided efficient, powerful statistical and optimization functionality.

Most participants utilized an IDE like Juno [10] or Visual Studio Code (VSCode) [15] 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 to allow users the full power of the tool support. For example, P3 takes advantage of the fact that, when a folder is opened, 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 be default" (P3). Additionally, P2 relies heavily on the fact that Julia's Vegalite [14] package is integrated into VSCode such that plots pop up in an integrated window and P9 used Jupyter Notebooks for diagnostics and real time work.

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) although they did know that it was possible.

Overall, 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.

4.1.3 Distinction Between Host Language and eDSL. Participants expressed confusion about the distinction between Mimi and Julia, sought clarity on the dividing lines, and desired intertwined documentation so they could learn the Julia concepts they needed for specific Mimi tasks. These outcomes appear to be side effects of following the design tenet 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" [34] and that embedded DSLs should be a "seamless extensions to the language" [34].

We have already detailed how the blurring of the GPL–eDSL line affects participants' understanding of error messages (Section 4.1.3), but this theme appears in a few other patterns.

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)

This blurred distinction also seemed to make it hard for participants to know where to look, or where to focus, for debugging and performance improvement. When P3 encountered a Julia parsing error, as described in detail in Section 4.1.5, 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).

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.

4.1.4 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 2.4). However, asking users to understand more host language features can make the eDSL less usable.

Participants themselves identified 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)

Package Development. Mimi documentation encourages formatting completed Mimi Models as Julia *Packages* as opposed to *Projects*. This design decision means that a user working with one of these completed models must understand how *developing a Package* [8] 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 usability 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 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 simplify 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)

Eventually these participants learned, from experience and 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 our session, P7 spent over 30 minutes trying to transition to the new suggested workflow. This included reading documentation and 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 to check and 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 the extra 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 [4, 7], questions on forums, and even custom programming environment tooling in environments like VSCode [5]. 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 process.

Multiple Dispatch. The Mimi implementation makes heavy use of Julia's support for *multiple dispatch* [12, 13]. The use of multiple dispatch allows Mimi to essentially overload Julia Base functions like `getindex`, `show`, `build`, or additional 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 "saavy with types" (P8), multiple dispatch errors—which rely on familiarity with the involved types and some understanding of how Julia dispatches based on types—represented a usability obstacle. Section 4.1.5 covered error messages and described 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 has had the knock on effect of exposing Mimi users to an apparently confusing category of type-centric 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 usability issues. For example, the eDSL somewhat protected participants from performance pitfalls, especially in potentially low-performance tasks like Monte Carlo Simulations [9]. Some programmers find Julia documentation thin [6, 16], but we did not identify this as a primary challenge for our participants, who had access to the Mimi documentation and forum (Section 4.2). The Julia community identifies confusion and frustration around subtyping logic, method specificity rules, and the verbosity of Union type declarations as usability barriers [16, 17]. 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 dive into 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 [16].

Summary. 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 less usable. On the other hand, Mimi successful shielded users from some common host language issues. This suggests designing for usable eDSLs may require understanding how implementation decisions affect the parts of the host language that users will be required to use and understand. This kind of understanding may help designers change implementations to shield users from known-complicated parts of the host language.

4.1.5 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 error messages are often confusing for users because they reflect the concepts and vocabulary of the host language instead of the DSL (Section 2.4).

Julia errors commonly refer to the Julia type system. If Julia cannot choose which function method to call based on the argument's run-time types, it will throw a `MethodError` signaling that there is no `Method` for the given Type(s). To a user new to the Julia type system, these errors may seem unintelligible: P9: "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)

For 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 tried to access the parameter values of `paramname` of component `compname` in *un-run* model `mymodel` by typing `values = mymodel[:compname, :paramname]` (brackets being 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 referred to using brackets, so they observed different places where they used this access method, 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, this was partially based on "seeing this error before" (P7). Without a comfort with the Julia type system and the errors associated with it, the common `MethodError` was difficult for participants to understand.

In another case, an error messages thrown by P3's small syntax error (including a 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, and will be discussed further in 4.1.6. This represents another case of error messages that reflect host language vocabulary impacting usability.

4.1.6 Host Language Syntax vs. eDSL Syntax. We see another set of issues when users *or programming tools* draw parallels between host language syntax and eDSL syntax. In cases where the syntax matches exactly, these kinds of parallels may help users pick up eDSL syntax. In cases where the syntax is subtly different, these kinds of parallels may produce difficult-to-diagnose issues.

P3 uncovered an unexpected syntax mismatch via using an automatic code formatter. Their automatic formatter puts 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.

Mimi's macro-implemented abstractions use syntax that resembles—but *does not exactly reproduce*—Julia function syntax. Some Mimi macros include elements that share syntax with Julia functions, but which are actually parsed in the macro instead of run directly as functions. For example, the following line creates a parameter `temperature`:

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

In this instance, the participant's formatter matched the syntax to a Julia function, and automatically inserted a semicolon:

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

While this part of the macro mirrors a Julia function call, the parsing logic in the macro does not handle a semicolon, 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, surface-level similarities coupled with subtle inconsistency between embedded DSL 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.

4.2 Community Engagement with eDSL-Specific Community

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

4.2.1 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 [2].

Of the ability to contact Mimi developers 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 that on the well-known platform Stack Overflow they avoided writing a question, 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).

4.2.2 Community Engagement Impact on Adoption of eDSLs. Participants also reported that community engagement was key for adoption. This pattern is known for GPLs [40, 50], 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, and/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)

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.

Summary. Our observations suggest eDSL users may prefer eDSL-specific forums and interaction with an active eDSL user community for satisfactory usability. 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 usability may be affected by access to their own specialized communities.

4.3 Data I/O and Host Language Support

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 it into a usable 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 I/O 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 usable Data I/O support from scratch.

Mimi represents integrated assessment models with a modular structure, using self-contained `Components` 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)

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)

The pain of transitioning data to and from disk, transitioning it between 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 having to read and write data. The embedded approach may support language designers in reducing unpleasant manual and error-prone data I/O operations for their users.

4.4 Existing Code and Templates

Participants used Mimi tutorials and fellow Mimi users' programs as starting points for their work. Section 4.2 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.

Copy-Paste-Modify. Many participants created a new program by finding whole programs or snippets of existing programs that they can adapt to meet their needs. P4 made several incremental modifications to an existing model. For each modification, they (1) looked through the existing model for examples of the function or syntax they wanted to use, (2) directly copy-and-pasted the example code to their desired location and, (3) 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)

If an 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. Alternatively, 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 [24, 27, 31, 48, 54, 59] suggests this behavior is prevalent across a variety of domains. This copy-paste-modify behavior also means that practices exhibited in early public code bases 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.

Tutorials as Templates. Many participants used tutorials in particular as a source of starter code. 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 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`.

Diverging from Templates. In cases where the desired code diverged too far 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. Their new component was of similar *conceptual* scope, but far more detailed in its representation of processes. P2 had to frequently scroll through the component to get a high-level understanding, and they frequently paused to search for a specific section of interest. Starting from scratch, this component may have been more manageable and readable if broken up into sub-components. While P2 noted that the component was long, even inconveniently long, they did not seem to consider breaking it up into subcomponents. We hypothesize this may reflect that they were following the structure of the existing model.

P7 augmented an existing IAM by adding the ability to calculate the social cost of hydrofluorocarbon. They first found the control flow section in the model that handled the existing list of gas options:

```
if gas == :CO2 # carbon dioxide
...
elseif gas == :N2O # nitrous oxide
...
elseif gas == :CH4 # methane
...
end
```

The blocks of code indicated with "`...`" for each gas are nearly identical. As a first pass, P7 simply extended the control flow with:

```
elseif gas == :HFC
...
```

While they "originally tried to just copy this [existing] style of adding a component" (P7), P7 soon realized that the model 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".

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.

5 DESIGN IMPLICATIONS AND DISCUSSION

Below we summarize implications for DSL designers, arranged to mirror the structure of Section 4, then follow with a discussion of design considerations and areas for future research.

5.1 Design Implications

Here we include a condensed summary of the design implications associated with our findings.

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

Errors expressed in host language concepts slowed down and confused participants. Designers should be aware of common cases in which users may be exposed to host language concepts in error messages. They may wish to consider catching and customizing these messages as much as possible. Designers should not assume that users will have a deep understanding of 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.

The usability and utility of an embedded DSL is closely tied to the host language's package or library ecosystem and the ease of integration with other tooling. When choosing the host language for an eDSL, designers may wish to consider what functionality the eDSL will automatically inherit from the package ecosystem, and if the host language works well with the tooling and programming environments that domain users will need. The fact that "features come automatically and for free" [44] from a host language is a key *advantage* of the embedded approach, but 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.

The blurred distinction between host language and embedded language confused and hindered participants. Seamless integration of the embedded DSL into the host language is a popular design guideline for eDSLS, and it may have advantages for both the user and the developer. That said, our findings indicate that designers should anticipate that confusion about the difference between the host language and the embedded DSL 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.

The design of the embedded DSL controls which part of the host language participants need to 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 usability 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 Participants engaged primarily with eDSL-specific resources, and reported these resources drove adoption. Existing research indicates "developer preferences...are shaped by factors extrinsic to the language" [50]. While eDSL design guidelines emphasize decisions about intrinsic features of an eDSL, designers should expect significant effects on adoption and usability 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 Participants were sensitive to the time cost and error proneness of data I/O between different languages and file formats. Embedded DSLs intended to perform domain-specific tasks involving data can help users avoid tedious and error-prone data I/O work by prioritizing functionality that reduces or simplifies data I/O. An embedded approach can make this easier for eDSL designers by providing the underlying, often complex, features with which they can build custom lightweight data I/O abstractions.

5.1.4 Participants relied heavily on existing code and avoided starting from scratch, a process supported by some eDSL features which also results in propagating existing syntax and practices. Designers should be aware that "examples will be the archetypes for thousands of programs" [24], so they should expect 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, users still want examples of domain-specific usage. Dedicating space to code examples in documentation, or adding explicit templates for users, may support eDSL usability in the long run. Writing tutorials based on the understanding that programs will be used as templates may also support usability. Furthermore, eDSL designers may wish to provide abstractions that support users in building upon each other's work, or provide other resources that explicitly support or guide this work practice.

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 usability suffers. Instead, their 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 inherent or historic community patterns.

The climate change research community is often placed at the forefront of contentious debates with strong policy implications, leading to prioritization of replicable workflows and vetted, trustworthy templates and boilerplate code [25, 32]. 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 4.2). 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 4.3). Participants in this domain 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). Unfortunately 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 4.1.1).

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

Asking users of a small, embedded DSL to engage with an array of features from the host language may decrease usability, 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 (see Section 2.3), designers may want to consider *shielding* users from parts of the host language that are complex and could reduce usability. At least some subset of host language features will be exposed to users, and in this case designers may wish to follow requests like that of P6 in Section 4.1.4 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 Mimi's designers and the subset of Mimi users who were most comfortable with programming, while largely hindering less comfortable programmers. Powerful Julia features—high-performance numerical computing, emphasis on ease of metaprogramming, and multiple dispatch—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 4.1.5). 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 reduced usability (Section 4.1.1).

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 4.1.2) 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 2), and while we do find support for many of them in this work, some were not supported by our observations. For example, Sections 4.1.3 and 4.1.5 detail how blurring the distinction between host language constructs and eDSL constructs can obstruct novice Julia users. 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. Similarly, the literature often encourages designers to give eDSL users access to the full power of the underlying host language. Our findings do not fully contradict this design tenet, but they do temper it with evidence of the usability consequences associated with revealing host language complexities (Section 4.1.4 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 some evidence that some existing guidelines may unintentionally reduce eDSL usability.

6 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 four promising directions in particular. For each of these directions, future work could, for example: (1) explore how these patterns vary across choice of host language, choice of embedded language, and choice of target audience or (2) 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—and DSLs more broadly—can meet the programming needs of the large and varied body of domain-specific experts seeking automation support.

REFERENCES

[1] 2016. *Mimi Framework.* https://github.com/mimiframework

[2] 2019. *Mimi Framework Forum.* https://forum.mimiframework.org

[3] 2019. *The Serious Downsides To The Julia Language In 1.0.3.* https://towardsdatascience.com/the-serious-downsides-to-the-julia-language-in-1-0-3-e295bc4b4755

[4] 2020. *Developing your Julia package.* https://medium.com/coffee-in-a-klein-bottle/developing-your-julia-package-682c1d309507

[5] 2020. *Using VS Code for Julia development.* https://www.youtube.com/watch?v=IdhnP00Y1Ks

[6] 2021. *The Depressing Challenges Facing The Julia Programming Language In 2021.* https://towardsdatascience.com/the-depressing-challenges-facing-the-julia-programming-language-in-2021-34c748968ab7

[7] 2021. *How to Create Software Packages with Julia Language.* https://jaantollander.com/post/how-to-create-software-packages-with-julia-language/

[8] 2022. *Developing packages.* https://pkgdocs.julialang.org/v1/managing-packages/#developing

[9] 2022. *How to optimise Julia code: A practical guide.* https://viralinstruction.com/posts/optimise/

[10] 2022. *Juno.* https://junolab.org

[11] 2022. *Jupyter.* https://jupyter.org

[12] 2022. *Methods.* https://docs.julialang.org/en/v1/manual/methods/

[13] 2022. *Multiple dispatch.* https://en.wikipedia.org/wiki/Multiple_dispatch

[14] 2022. *VegaLite.jl.* https://www.queryverse.org/VegaLite.jl/stable/

[15] 2022. *Visual Studio Code.* https://code.visualstudio.com

[16] 2022. *What's Bad About Julia.* https://www.youtube.com/watch?v=TPuJsgyu87U

[17] 2022. *What's bad about Julia?* https://viralinstruction.com/posts/badjulia/

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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).*

[23] 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.

[24] 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.

[25] 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).

[26] 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.

[27] 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

[28] 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.

[29] 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.

[30] Steve M Easterbrook and Timothy C Johns. 2009. Engineering the software for understanding climate change. *Computing in science & engineering* 11, 6 (2009), 65–74.

[31] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing.* MIT Press.

[32] 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.

[33] Martin Fowler. 2010. *Domain-specific languages.* Pearson Education.

[34] 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.

[35] Pedro Gabriel, Miguel Goulao, and Vasco Amaral. 2011. Do software languages engineers evaluate their languages? *arXiv preprint arXiv:1109.6794* (2011).

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] John Hosking, Nikolay Mehandjiev, and John Grundy. 2008. A domain specific visual language for design and coordination of supply networks. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 109–112.

[43] Paul Hudak. 1998. Modular domain specific languages and tools. In *Proceedings. Fifth international conference on software reuse (Cat. No. 98TB100203)*. IEEE, 134–142.

[44] Samuel N Kamin. 1998. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science* 14 (1998), 149–168.

[45] 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.

[46] 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.

[47] 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.

[48] Shriram Krishnamurthi and Tim Nelson. 2019. The human in formal methods. In *International Symposium on Formal Methods*. Springer, 3–10.

[49] 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.

[50] 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.

[51] Engineering National Academies of Sciences, Medicine, et al. 2017. *Valuing climate damages: updating estimation of the social cost of carbon dioxide*. National Academies Press.

[52] 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.

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

[54] 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.

[55] 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.

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

[57] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. 2010. Embedding languages without breaking tools. In *European Conference on Object-Oriented Programming*. Springer, 380–404.

[58] Daniel A Sadilek. 2007. Prototyping domain-specific languages for wireless sensor networks. In *Proc. of the 4th Int. Workshop on Software Language Engineering*. Citeseer, 76–91.

[59] 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.

[60] 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

[61] Hadley Wickham. 2016. Data analysis. In *ggplot2*. Springer, 189–201.

[62] Dean N Williams. 2014. Visualization and analysis tools for ultrascale climate data. *Eos, Transactions American Geophysical Union* 95, 42 (2014), 377–378.