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

Lisa Rennels
Sarah Chasins

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-82
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-82.html

May 12, 2022

Acknowledgement

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: Professor David Anthoff, Professor Koushik Sen, the PLAIT lab at UC Berkeley.

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

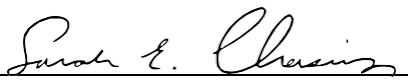
by Lisa Rennels

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

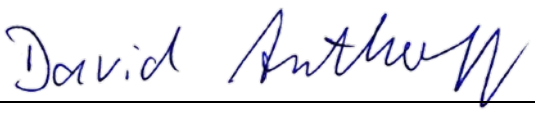
Committee:



Professor Sarah Chasins
Research Advisor

05/12/2022

* * * * * * *



Professor David Anthoff
Second Reader

05/12/2022

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

Software and data analytics are increasingly central to the work of domain experts across myriad fields, many of which lie entirely outside of traditional computer science. This trend makes domain specific languages (DSL) a valuable and popular tool for software engineers to support to research and work in particular domains, including the earth sciences. In particular, embedded DSLs are an approach to efficiently developing powerful and rich languages and tools for niche domains. Improving understanding of how embedded DSL programmers use the language including usage patterns, pain points, and the pros and cons of various intrinsic and extrinsic language features, is key to informing future design that prioritizes usability. We conducted a contextual inquiry including observation and short semi-structured interviews with the users of an embedded DSL, Mimi, written as a platform for climate economics domain experts in the integrated assessment modeling field. We then used thematic analysis to build up a hierarchy of primary themes from the audiovisual data. These including five primary findings, each with a specific implication for future embedded DSL design. These findings include that the host language of an embedded DSL is very consequential for the user, not just the designer, and that personal engagement with community is crucial to attracting and retaining users.

CCS Concepts: • **Human-centered computing** → **HCI design and evaluation methods**; • **Software and its engineering** → **Designing software**.

Additional Key Words and Phrases: embedded domain specific language, usability analysis, contextual inquiry, need-finding

1 INTRODUCTION

Software is an integral tool for research and analysis in myriad domains, including highly specialized areas with no formal relation to computer science. Experts require programming languages and tools that allow them to carry out domain specific tasks efficiently and correctly. These domain experts share common problem formulations, vocabulary, and knowledge and not necessarily formally trained in computer science. This trend is apparent in the earth sciences, and particularly in the climate change domain [2, 13, 14, 34].

Domain specific languages (DSL) are a long-standing and popular strategy for meeting the needs of domain users [16]. Embedded domain specific languages leverage the constructs of a host general purposes language (GPL) and add a (relatively) thin layer of domain-specific elements specialized for a particular community of users and their needs [25]. These embedded DSLs can also be considered libraries for the host language. This embedded approach, in contrast to the development of an entirely new language, is efficient for the designer and provides a rich, productive tool for the end-user, although it inevitably comes with limitations and complications introduced by the host language [11, 17, 23, 25].

Empirical usability analysis empowers software engineers to improve upon an individual domain specific language and advance the field's understanding of best practices for user-centered design of DSLs [27]. Such work is crucial to ensuring that DSLs do, in fact, provide value to the end users they target [6, 12, 31]. In the present contextual inquiry,

we employ both observation and semi-structured interviews to gain knowledge about (1) the uses, workflow, pain points, and desired improvements of an existing embedded DSL in the climate change economics domain and (2) gain generalizable insights about the design of embedded DSLs.

The *primary findings* of this paper are:

(1) Users rely heavily on existing code and avoid starting from scratch, resulting in propagation of existing syntax and practices.

(2) The host language of an embedded DSL is very consequential for the user, not just the designer.

(3) Mapping to a mental model is highly valued by users and is key for efficient use of an embedded DSL.

(4) Personal engagement with community is crucial to attracting and retaining users.

(5) Users are highly averse to the time cost and error proneness of data I/O between different languages and file formats.

The rest of this paper is organized as follows. Section 2 introduces the Mimi language and the Integrated Assessment Modeling domain. Section 3 details our study design, including recruitment, participants, session protocol, informed consent, and analysis. Section 4 discusses our findings in detail, while Sections 5 through 7 summarize related work, study limitations and next steps, and conclusions, respectively.

2 MIMI AND INTEGRATED ASSESSMENT MODELS

Integrated assessment models (IAM) are used to calculate the social cost of carbon dioxide (SC-CO_2), an economic metric employed extensively to inform a wide 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 setting the value of carbon taxes in federal 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 William Nordhaus, who pioneered these macroeconomic models in the early 1980s [29].

2.1 Motivation for Mimi

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, leading to an all-too-often cursory take, or outright omission, of such work. 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 remedy these problems [28].

Mimi (https://www.mimiframework.org) is a macro-based domain specific language embedded within the Julia general purpose language which fills this need for a uniform, integrated software platform that co-locates the body of current and in-development IAMs. Mimi uses a readable, easy-to-use API to streamline model creation and editing, encourages collaboration with a modular approach, and provides infrastructure for sensitivity analysis and results visualization

while leveraging the features of Julia including readable syntax, multiple dispatch, dynamic typing, high performance numerical computing, and advanced environment/package management support [7].

2.2 Design Decisions

Pairs of key motivation or need and the resulting design decisions are listed below. Several come directly into play in our findings.

- computationally fast → macro based embedded domain specific language in Julia
- simple enough for novice programmers and works with the pre-existing ecosystem of models → a readable, easy-to-use API that leverages the full power of Julia including significant employment of multiple dispatch
- enables a modular work style for distributed, loosely coordinated teams → use open-source host language and the Julia registry
- creates a transparent framework enabling easy replication of computational experiments → leveraging of packages and project environments

In addition, Mimi heavily leverages (1) custom, mutable composite data types that Julia defines as "a collection of named fields, an instance of which can be treated as a single value" (docs), and are also known as structs, objects, or records in other languages, and (2) macros.

3 BACKGROUND AND RELATED WORK

The following discussion of background and related work is separated into three major categories: (1) earth science and climate change, (2) embedded DSLs and usability, and (3) language adoption and feature importance.

3.1 Earth Sciences and Climate Change

Increases in computing power combined with an explosion of data collection and analytics places software at the forefront of cutting-edge research in the earth sciences, and particularly the climate change, domains [2, 34]. Along with this phenomenon comes a focus on open source software, reproducibility, and best practices within the disciplines [10, 15, 18]. Particular to the climate change domain, Easterbrook presents the "Grand Software Challenge" and maps out areas for contribution and research by software researchers and engineers [13].

A growing body of literature presents the case for contributions to this domain from the software engineering community, however there is a paucity of work attempting to rigorously examine the gaps and needs (1) with empirical methods and (2) as an opportunity for research on and application of domain specific languages. Easterbrook's ethnographic study of climate scientists building and working with extensive, advanced code bases that inform climate forecasts, climate change research, and policy making is a key piece of empirical research on this topic [14]. Our work adds to the empirical literature and uses it to advance the understanding of the usability and usability centered design of embedded domain specific languages.

3.2 Embedded DSLs and Usability

A long standing body of literature on the design of embedded DSLs, and the advantages and disadvantages of the embedded approach, informs our study. Hudak emphasizes the advantages of using the embedded approach to domain

specific language design [22, 23]. Mernik et al.'s piece on developing domain-specific languages presents an overarching review answering the question of when, and how, one should develop a DSL [25]. The authors state that DSLs "trade generality for expressiveness in a limited domain" and enumerate important design pattern decisions like language exploitation, where the DSL takes advantage of an existing GPL, versus language invention. The former design pattern is at the root of *embedded* domain specific languages, and the authors go on to carefully enumerate the advantages and disadvantages of the embedded approach. We review a subset of these pros and cons below, augmenting the findings of Mernik et al. with several additional papers.

- *Advantage: Minimal Design Effort for a Powerful Language* Embedded domain-specific languages allow designers to take full advantage of the features of the underlying GPL, allowing for an efficient development process resulting in a powerful language [25]. As emphasized by Kamin, "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" [24].

- *Advantage: Existing Infrastructure and Tools* Not only does the host GPL provide language features for the embedded language, but it also allows users to take full advantage of additional infrastructure and tooling, such as "development and debugging environments: editors, debuggers, tracers, profilers, etc" [25]. Renggli et al. describe that cases where this integration is not smooth, such that "editors, compilers, and debuggers are either unaware of the extensions, or must be adapted at a non-trivial cost", pose serious difficulties for users and suggest an approach to ensure extensions will function.

- *Disadvantage: Constraints on Syntax.* Adopting a given host language means that the DSL syntax will be highly constrained by the choices made by the GPL developers [25]. Freeman and Pryce describe the experience of writing an embedded DSL in Java, and find that while there was value to taking advantage of the GPL, "the conventions of the host language are unlikely to apply to an EDSL, given that the motivation for writing an EDSL is to overcome limitations in the host. To make the EDSL readable, it may need to break conventions such as capitalisation, formatting, and naming for classes and methods" [17].

- *Disadvantage: Confusing Error Reporting* A drawback of the embedded approach is that error messages produced by the GPL infrastructure are often confusing for users since they are expressed "in terms of host language concepts instead of DSL concepts" [25]. Kamin warns of these "incomprehensible" error messages, and Freeman and Pryce discuss adding features such that error message reporting is readable for users of the DSL [17, 24].

- *Disadvantage: Confusing Results from Overloading* While the ability to overload operators of a host language may be a key part of the advantage of using an embedded language, it can also cause substantial confusion when the syntax of the DSL diverges from that of the host language [25]. As Freeman and Pryce worked on a domain-specific language, they found that "there is a balance to be struck when overloading operators or abusing keywords to create an embedded language. If in doubt, be conservative and define operators and keywords to have as close a meaning in the embedded language to that which they have in the host language or its standard library" [17].

We provide further empirical evidence and nuance to many of these claims, discovering them through a bottom-up inductive, semantic thematic analysis and later observing how findings map back to the literature. In particular, we add an empirical study in a domain rarely studied in this context.

Empirical usability studies have grown in popularity recently, and are valuable insights for this work. Barivsic et al. perform an empirical study of 128 teenagers and their use of the Gyro Creator Language DSL, as compared to other similar robotics languages for teens new to computer science, and evaluate it across several categories of usability [5]. Patnaik et al. look to the Green and Smith's ten principles for usable cryptography libraries to examine the usability, or lack there of, of crypto libraries [19, 30]. The authors perform a thematic analysis over Stack Overflow comments on 7 different crypto libraries, discovering sixteen themes and mapping them back to the principles of Green and Smith.

3.2.1 Frameworks. Clarke and Becker adapt the *cognitive dimensions* framework to analyze usability of class libraries and incorporate this framework into class library design and analysis via empirical usability studies at Microsoft [8, 12]. Similarly, Albuquerque et al. suggest a set of metrics derived from the cognitive dimensions to evaluate DSLs and perform an empirical study using these measures to compare two textual DSLs [1]. In another framework approach, Poltronieri et al. present the Usability Evaluation of Domain-Specific Languages (Usa-DSL) framework [31]. This eleven step process is rooted in Human Computer Interaction (HCI) concepts, and the paper includes a detailed description of the framework as well the results of a focus group study which examines its usefulness. Finally, Barisic et al. propose the Usability Software Engineering Modelling Environment (USE-ME) framework to iteratively develop DSLs with considerations about usability [3]. Their work builds on a large body of work produced by a cohort of researchers on DSLs and usability, including both theoretical suggestions, evaluations of proposed frameworks, and case studies [4, 6].

Our work uses these frameworks as context and grounding for our findings, although we decide to use an inductive approach to thematic analysis, allowing our observations to suggest our themes without a top-down application of a framework.

3.3 Adoption and Feature Importance

The existing literature on language adoption and feature importance for developers offers supporting insights for the present work. This is a fairly large literature, so we do not offer an exhaustive review here, but cite a few examples which relate specifically to our findings.

Head emphasizes the increasing importance of social health cues, defining a package to be "socially healthy when a developer can reliably get helpful answers from a package's forums, mailing lists, and other communications channels" and conducts a user study to examine how users assess social health of a package [20]. Myerovich and Rabkin perform surveys on large datasets to examine programming language adoption, including how developers choose what language to use, language acquisition, and beliefs about languages [26]. Numerous findings in this piece are specifically relevant our findings, including that (1) users tend to value extrinsic language features over intrinsic ones and (2) DSLs are less popular overall as compared with GPLs, but within certain niches can prove more popular.

We add to this literature by using contextual inquiry to observe users, and in our focus on domain specific languages.

4 STUDY DESIGN

For this study we conducted a contextual inquiry with nine participants. Contextual inquiry is a field study technique rooted in user centered design during which the researcher performs in-depth observation of users with minimal interruption, seeking to as best as possible observe them work as the would naturally on their own work [21]. We performed an inductive, semantic thematic analysis of the audiovisual data. For this analysis we used open coding to

attach short descriptive labels, or codes, to segments of the videos and then iteratively built up a hierarchy of observed themes.

Recruitment. We recruited nine participants via snowball sampling starting with (1) recruitment emails to individuals and groups and (2) a recruitment post to the public user forum.

Participants. The nine participants represent a variety of industries, years of experience in the climate change domain, and levels of formal and information education. The most reported programming languages reported used in the last two weeks (other than Julia and Mimi) are R and Python. Figure 1 details key information about these participants obtained with a survey.

ID	Age	Employment	Yrs in Climate Change Domain	Setting for Climate Change Domain and Mimi.jl Work	Programming Education (formal and informal)	PL and Tools (used in last 2 weeks)
P1	30	PhD Candidate	7	academic research	- Numerical modeling courses during BSc - Self-taught during graduate school	R
P2	32	Post Doctoral Researcher	10	academic research	- MSc in Science and Executive Engineering - Informal training with Lisa Rennels and David Anthoff	Julia (including Mimi)
P3	28	Assistant Professor of Civil and Environmental Engineering	9	academic research	- 1 course in undergraduate - Several years of informal self study - Various online tutorials and parts of online classes	Python, Julia (Jupyter)
P4	37	Economist with Federal Government	2	academic research government	- 100% informal as a complimentary necessity - Research needs and interests required software, software required programming, and these resources or training were not in the curricula	R, Stata, Julia, Mimi, LaTeX, Python
P5	40	Assistant Professor and Computational Scientist	8	academic research	- Formal education and work in C++, Scheme	R, Python, Julia, Javascript
P6	59	Associate Professor in Department of Computational & Data Sciences	30	academic research	- Formal coursework in Fortran and NetLogo - Self-taught Pascal, SAS, Julia, Python, R, Visual Basic	NetLogo, Julia,
P7	22	Research Assistant for US Environmental Protection Agency	1	government research	- Statistics undergraduate degree (R) - Online courses (Python)	R, Julia, Python
P8	28	Research Associate at NGO	1	NGO	- Data science course as part of grad school - R-based ecological dynamics course	R, Julia
P9	33	Assistant Professor of Mathematical Sciences	11	academic research	- Introductory computer science in undergrad - Learning on the job and teaching Computer/Data Science	Julia, Python, R, Excel, GitHub, Google Colab, Jupyter notebook

Fig. 1. Key information about study participants.

Session Protocol. Each session began with a short introduction to the study and reaffirmation of informed consent. We then conducted an approximately 50-minute contextual inquiry session, during which participants worked on an existing 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 short semi-structured interview of approximately 10 minutes.

Informed Consent. Participants signed a consent form prior to the session in accordance with the UC Berkeley Institutional Review Board. Participation was voluntary and participants received small monetary compensation.

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 and then iteratively build up a hierarchy of observed themes.

5 FINDINGS

In this section we present the key findings from our contextual inquiry.

5.1 Users rely heavily on existing code and avoid starting from scratch, resulting in propagation of existing syntax and practices.

A core tenet of the Mimi embedded DSL is supporting research collaboration, and the main goal of most participants is to modify existing models. For example, P7 adds new greenhouse gases to calculate their respective social costs (ie. SC-HFC1), P4 modifies a model by adding settings options for alternative parameterizations, and P2 adds new feedbacks between existing components. While use cases show that support of collaboration is thus far successful, we note that (1) the use of existing code propagates legacy code practices and patterns from the original template (be it a tutorial or just another user's code) and (2) users avoid starting from scratch and frequently directly copy and paste existing code or examples to start from.

Example 1

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 script. Referencing documentation as needed, they modified the script to match their model. While the resulting code was functional, it was bloated with unnecessary code. The `@defmcs` macro allows users to either (1) create a random variable with a name and distribution and then subsequently attach a Parameter to that random variable:

```
MyParam = RV1
RV1 = Normal(0,1)
```

or (2) use a shortcut to do step (1) in one line:

```
MyParam = Normal(0,1).
```

The tutorial was an "everything-but-the-kitchen-sink" tutorial, including many possible functions within one example instead of keeping to a terse representation of what a user might *actually* want to do, so it included both implementations:

```
RV1 = Normal(0,1)
MyParam = Normal(0,1)
```

The first line creates a random variable `RV1` and the second line creates *another* (anonymous with a structured name under the hood) random variable and attaches `MyParam` to that random variable. Hence the first line serves no purpose. Copying the tutorial resulted in P3 ending up with similar repetitiveness in their code, and they expressed confusion at why the first line was included. Relevantly, Bloch warn that "example code should be exemplary. If an API is used widely, its examples will be the archetypes for thousands of programs" [9].

Example 2

P4 made several incremental modifications to an in-progress model, and for each one their process was to (1) look through the existing model for where the function they wanted to use, or action they wanted to take, had already occurred (2) directly copy and paste the the code to the new location (3) modify as needed. We note similar behavior in P1, P2, and P7. In fact, P1 explicitly walked us through their initial workflow, showing that they "went to the main components ... just the ones that I wanted to change [and] copy and pasted what was within each component [into a new file] and then added the new parameters that I wanted." In this case, they started with an existing model and then copy and pasted the components they wanted to modify into new files, made the modifications, and swapped them into the model to replace the older versions.

The value of ability to extend code in this way is emphasized in Meyerovich and Rabkin (2016) as highly important to selection of language by users [26]. Also notable from the literature, Thayer, Chasins, and Ko (2021) examine the role example code and templates within API knowledge, use, and learning [33].

Implication. As perhaps stated best by Bloch, designers should be aware that "examples will be the archetypes for thousands of programs", and that if they don't provide examples and templates, they cede space to less vetted sources like the code of existing users. New users will seek out starter scripts and templates even if they don't see them in official sources, which may cause unexpected or outdated language usage propagate over time. Dedicating time to code examples in documentation, or adding functionality to create and populate templates for users, may serve designers well in the long run.

If possible we should add an Example 3 of a specific instance where the boilerplate code used by these participants was outdated in syntax, or used practice that deviated from what a tutorial or Mimi expert would have advise this occurring, perhaps P7 adding a gas with copy/paste but then realizing this method won't work, although ideally a case where the specific syntax is outdated or incorrect.

5.2 The host language of an embedded DSL is very consequential for the user, not just the designer.

One of the most dominant themes that came out of our work was the multitude of ways that the choice of host language (Julia) affected not only the work of the embedded DSL developers, but the *usability of the DSL from the perspective of the users themselves*. In one telling quote, P9 stated that "my frustration with Julia and my frustration with Mimi run together a lot". Many of these findings are recognized by the literature reviewed in the Embedded DSLs and Usability: Embedded DSLs section of Related Work above.

This general theme ties together a large portion of the observations that constitute our analysis, so we will spend a substantial segment of this Findings section on this topic. For clarity, we further subdivide this theme into five distinct but related subthemes.

(1) Errors expressed in host language concepts slow down and confuse users.

(2) When an embedded DSL designer chooses a host language they also choose the package management system users will have to use.

(3) The usability of an embedded DSL is closely tied to the host language's package or library ecosystem and the ease of integration with other tooling.

(4) The blurred distinction between host language and embedded language creates both curiosity and confusion for users.

(5) The design of the embedded DSL controls which part of the host language users need to understand.

5.2.1 Errors expressed in host language concepts slow down and confuse users. Error messages returned from Mimi 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. However, when users encounter errors thrown directly by the host language, they hit pain points and a slow debugging processes caused by unintelligible error messages using the concepts of the host language. The literature on embedded DSLs discusses the problem of error message context at length, recognizing that error messages are often confusing, even incomprehensible for novice users, because they reflect the concepts and vocabulary of the host language as opposed to the domain specific language [17, 24, 25]. Relatedly to some of these examples below, Freeman and Pryce find "there is a balance to be struck when overloading operators or abusing keywords to create an embedded language" [17].

Example 1

P9 expressed that "The Mimi API errors in a similar way to the way Julia errors, such that the error message for someone who isn't saavy in all the types like *Abstract Bool* can be a little bit like morse code sometimes". A typical Julia error in this context may indicate that there is no *Method* for a given Type, but to someone new to the type system of Julia, and not asked to explicitly learn it for Mimi, these errors may seem "like morse code"!

Example 2

P6 omits a colon and passes `name3` to a function instead of `:name3`. The former is interpreted as a variable to be found in the namespace, whereas the latter is a `Symbol` type in Julia, similar to a `String`. When they attempts to run the line, they see the error:

```
LoadError: LoadError: UndefVarError: name3 not defined
```

They said "I see I made a mistake but I don't know why" and carefully scanned their code before trying to run it again without making any changes. Upon a second pass looking for all uses of `name3` they see the missing colon, correct it, and say "There's that silly Julia thing again" indicating they had issues with the foreign `Symbol` type before.

Example 3

When P7 tried access parameter values for parameter `param` in component `comp` from model `m` that had not yet been run using `getindex(m::Model, comp::Symbol, param::Symbol)`, they got the following error.

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

This occurs because a call to `getindex` on an *unrun* model dispatches to the `ModelInstance` field, equivalent to calling `getindex` on `Model.ModelInstance`, and that `Model.ModelInstance` is set to a placeholder value of `nothing` (Julia type `Nothing`) until a model is built at runtime. So when they type `m[:compname, :paramname]`, Mimi calls

`getindex(m.ModelInstance, :compname, :paramname)` and since `m.ModelInstance = nothing`, they get the Method Error above.

P7 noted that the error message "said something about getindex" so they looked through uses of `getindex` in the code they had added, including above and below the problematic line of code, for several minutes. After this trial and error, they finally matched the error message to the fact that the `Nothing` referred to the `m` and the `comp` and `param` being of type `Symbol` and concluded that the model needed to have been run first. This was partially based on "seeing this error before", which cued consideration of the side effects of running a `Model`. They corrected to

```
run(m)
m[:comp, :param]
```

Example 4

P3 used a automatic formatter that puts a semicolon between named and unnamed arguments in a Julia function for clarity (ie. `myfunc(2, option=false)` becomes `myfunc(2; option=false)`. This is standard recommended practice for syntax in Julia. The Mimi `@defcomp` macro includes commands with the same syntax as a Julia function, but which are actually parsed into several different calls within the macro instead of run directly as functions. For example, the following line creates a parameter `temperature` within the component definition macro `@defcomp`.

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

which automatically formatted to

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

While this part of the macro was intended to appear like a function call, the parsing logic in the macro does not expect to see a semicolon, thus adding a semicolon to the line causes an error. Unfortunately, this meant that the error observed by P3 was a low-level parsing error including the text

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

The user spent a full ten minutes trying to solve this problem, seeking help from online documentation and commenting out different lines to try to isolate the error. They in the end were able to pattern match to existing functioning code and online examples to recognize the extraneous semicolon being added did not match any examples or working code. They was not able to understand exactly why the semicolon was an issue, but were able to solve the problem. This very subtle inconsistency between embedded DSL syntax and host language syntax is already confusing for users, but made extremely painful to diagnose because it resulted in an error message about parsing that did not match user concepts or knowledge.

Implication. Designers should be aware of common cases when users may be exposed to host language context in error message, and consider catching and customizing these messages as much as possible. Furthermore, designers should not assume that users will have a deep understanding of host language constructs beyond what they are exposed to in the embedded language, and act accordingly when designing error messages to aid debugging.

5.2.2 When an embedded DSL designer chooses a host language they also choose the package management system users will have to use. Since Mimi is embedded in Julia, users must learn and engage with the Julia package management system. A Julia environment is defined by two configuration files, *Project.toml* and *Manifest.toml*. The former lists the packages to be loaded into the Environment and compatibility bounds, and the latter pins down exact package versions of this list and all its dependencies. The environment can be activated and instantiated (once) on any machine to perfectly replicate a working environment for a Project.

When asked about use of environments, P3 stated that they "love the Julia environments" and that it took some effort to learn. In fact, "a lot of [their] questions about how to do things are *workflow* as much as anything else". P7 also emphasized the learning curve, stating that "it was quite difficult and frustrating because I didn't understand how all the dependencies and environments didn't work. As I've gotten better with Mimi things have gotten less frustrating."

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 don't know why", indicating that double checking their environment settings is a helpful and common part of their workflow. They also integrate the `Pkg` package into their scripts, including

```
Pkg.activate("source_folder_name")
Pkg.instantiate()
```

This ensures they have activated the intended environment to run their scripts and work on their project. P9 does the same, with the comment next to that line stating *Activate the project and make sure all packages are installed*, showing they understand the use case and intent.

P8 started a fresh project with the default (empty) environment. They quickly realized that they had a pre-existing defined environment that would work well for the given quick experiment they wanted to run, and seamlessly activated that one so as to avoid the time of re-adding packages to new environment.

P6 spent the session working on tutorials, and had defined an explicit `MimiTutorials` environment to work in. When they began working on a new tutorial, they received an error that they had not downloaded a specific package, and quickly diagnosed the issue, typed `add Distributions` to add the Distributions package to the environment.

In contrast, P1 and P2 do not seem to engage actively with environment infrastructure, though the way it runs in the background for them seems to be unproblematic, and they use package management to add and use packages without trouble.

Implication.
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 this will be required to use the package management system of the host language, which may or may not be intuitive for them.

This is a bit thin but there is data to back it up. It would be good to go back through and carefully check (1) who uses environments actively and who does not (2) how they use them. We can use visual cues from the VSCode window even if it isn't discussed explicitly.

5.2.3 The usability of an embedded DSL is closely tied to the host language's package or library ecosystem and the ease of integration with other tooling. The user experience is substantially affected by the availability and stability of complementary Julia packages, as well as their ability to integrate into the larger ecosystem of traditional tools and infrastructure like Github and IDEs of choice.

In terms of the richness of the package ecosystems, several users expressed a desire for richer plotting packages in Julia. P2 said that "Julia is pretty new, Vegalite [the recommended plotting package] is pretty new, so there are some limitations". In addition, P1 and P4 explicitly step out of Julia and use R's *ggplot* to do plotting. This incurs the cost of data I/O as described in section 4.5. In addition, since Julia's packages are often still quickly evolving, P2 states that it "has been annoying" to have to rewrite their code to match syntax as packages release new versions.

P5 is a fairly advanced Mimi user, and they have integrated Mimi into new packages with Julia optimization and linear programming which has been "working quite well" for them. This integration has been key to their work, and also leverages the strengths of the Julia host language in numerical programming and performance.

A majority of users utilize a powerful IDE like Juno or VSCode to do their work in addition to either integrated Github functionality in the IDE, or a Github Desktop Application. P8 uses Jupyter Notebooks for diagnostics and real time work. The host language, Julia, is well integrated into these tools and Mimi can piggyback 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 and says "I like that the VSCode extension will do that be default". Similarly, P2 relies heavily on the fact that Julia's Vegalite package is integrated into VSCode such that plots pop up in an integrated window.

When P3 encountered a Julia parsing error, as described in detail in *section 5.2.1* above, they tried to use the Julia inline REPL `help?>` functionality to search `help?> Mimi.Parameter` and when that didn't work `help?> Parameter`, but they were unable to find any documentation. This is evidence for a place where a deeper integration into Julia documentation might have been helpful.

Mernik et al. highlight the light this advantage of the embedded approach, as opposed to writing a new DSL, while Renggli et al. descrube that cases where this integration is not smooth, such that "editors, compilers, and debuggers are either unaware of the extensions, or must be adapted at a non-trivial cost", pose serious difficulties for users [25, 32].

Implication. When choosing the host language to embed a DSL in, it is important to consider what functionality the DSL will automatically inherit from the package ecosystem, and if the host language works well with tooling the domain users will need to be able to use for smooth workflows. As Kamin emphasized, the fact that "features come automatically and for free" from a host language a key advantage of the embedded approach, so the designer should consider matching which and how many features they get to what a domain user will value most.

5.2.4 The blurred distinction between host language and embedded language creates both curiosity and confusion for users. Users note confusion about the line between Mimi and Julia, seek clarity on the topic, and desire intertwined documentation about the two such that they are educated about Julia as needed to support specific tasks in Mimi.

For example, P1 was curious enough about "what was Julia and what was Mimi" that they opened the source code files of Mimi locally to try to understand the distinction. They noted that they were new to both Julia and Mimi so they "didn't know what was very specific to Mimi".

This blurred distinction also makes it hard for users to know where to look, or where to focus, for debugging and improvement. For example, as mentioned in *section 5.2.2* above, P3 encountered an error they did not understand so tried to use the Julia `help?>` feature in the REPL to search for a the Mimi specific `Mimi.Parameter` type, and when that didn't work they tried `Parameter`. A few users also wondered aloud whether certain paints points, such as performance or speed issues, were a Mimi behavior or a Julia characteristic. For example, P4 and P7 were working on developing packages, which necessitated restarting the REPL after large updates. Each had to wait a 1-3 minutes upon each restart due to precompilation, and wonder aloud of this is something all Mimi development involves.

Implication. Designers should anticipate curiosity, and potential confusion, about the difference between the host language and the embedded DSL. This may be especially prevalent if users are new to *both* the host and the domain language.

This would be a good place to go back though the interviews looking specifically for places where participants may confuse the syntax of Mimi and Julia. We can also include more specific discussion of overloading etc. if it applies. We should similarly look for where this blur might be an *advantage or good thing* for users, or is it just helpful for us designers under the hood?

5.2.5 The design of the embedded DSL controls which part of the host language users need to understand. An embedded language will necessarily involve users in parts of the host domain language, such as the package management system as described above in *Section 2.5.4*, but the designer does have some choice over how much, and which other parts, of the language users will need to understand in order to utilize their library.

For example, in Mimi a majority of completed Models are Julia *packages*. While users can interact with and change a Model using the Mimi API and without directly editing the Model's root package, numerous users decide to edit or augment the Model in the source code due to the nature of their desired tasks and outcomes. At this point, to work efficiently the user must understand how developing a Package differs from working on a Project in Julia, and notably the implications for workflow.

For over 6 months, P4 and P7 developed existing packages using unconventional workflows as opposed to what Julia documentation would suggest as package development workflow. They experienced continuous frustrations of not seeing their changes to core package scripts picked up when using that package, and efforts to get changes picked up were unreliable and hard to understand. This was caused primarily by their system pointing to the package held in the General Registry, but changes being made to the locally developed version and not synced to the General Registry. Julia's recommended protocol would simplify this problem, but their workflow indicates a lack of understanding about how package development differs from other workflows, and thus they took a long time to realize a change in workflow would likely improve the experience.

P4 said that "one of the most frustrating things for me is why didn't my edits take" and spent "so many hours and days where my edits weren't accounted for ... obviously was my fault because something was wrong with my workflow". Even after shifting to the Julia recommended workflow, P4 was wary of changes not being picked up that "it is extremely unreliable as to what it is actually reading".

After over half a year, P4 and P7 learned from the Mimi forum and developer assistance there a more reliable and conventional workflow was available for package development. In our session, P7 spent over 30 minutes trying to transition to this new 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.

Once P4 transitioned to conventional workflows for package development, they found picking up changes to be reliable but slow. For an environment to pick up a change to a package in development, a user must either (1) kill and reopen the REPL or (2) use the `Revise` package, which allows many (but not all) classes of changes to be incorporated without killing the REPL. Not all users are aware of `Revise`, and P4 and P7 only learned about it from offhand conversations with Mimi developers.

P7 stated that "anything that requires me to close the terminal and reopen it is annoying because it takes such a long time to load everything up again so I refrain from closing VSCode as much as possible". They do note that `Revise` has been helpful. P7 has a relatively slow local machine, and each time they kill the REPL it can take a few minutes for the environment to be ready to work again. As said previously, although P4 now has a more consistent workflow they are wary of changes not being picked up, and they kill the REPL frequently because "it is extremely unreliable as to what it is actually reading". Each time they kill the REPL they may have to wait a few minutes for precompilation etc., and these processes are especially slow for this users as their system is low on memory and processing power. In addition, the tendency to kill the REPL as a catch all solution reveals that users may still not fully understand the reasoning behind the package workflow and its difference from project workflow.

Example 3

P3 did a large part of their results generation in Jupyter notebooks. By default, a Julia package is stored in a hidden folder in `.julia/dev/pkg_name.jl`. As P3 points out, "you can't put a Jupyter notebook in a hidden folder and that's where I just quit" although they do know that it's possible. In this way, setting up a Mimi model as a package added a barrier to taking advantage of tooling (*Section 5.2.2*.

Implication. An embedded DSL designer has some control over which parts of the host language users will have to understand and use in order to use the DSL. The decision to add a new host language feature to the list should not be taken lightly, as it has direct consequences for the complexity of using the DSL and can create barriers for users. Relatedly, designers should consider what kind of documentation and support the host language provides for a given feature, and how much additional instruction the DSL designer will need to provide to keep things smooth for users, before bringing it in to the embedded DSL.

Right now the only examples in this section are about the package development part of the host language, but I will return to the videos to try to find new examples.

5.3 Closeness of mapping to a mental model is a high priority for users.

One clearly successful aspect of the Mimi embedded DSL is the "closeness of representation to domain", or *Closeness of Mapping* in the Blackwell and Green Cognitive Dimensions of Notation's terminology [8]. The ability to carefully match domain concepts to their representations in language constructs is a key advantage of DSLs [25, 31].

P1 stated that Mimi "fit with a schematic model that I had in my mind ... it's really how I draw the integrated assessment models ... [the scheme in my head] is how [Mimi] is built". Similarly, P2 said that "I'm really happy that I can manipulate this whole thing box by box", and then showed a presentation they gave at a recent job talk, and how the schematic they draws maps perfectly to the Mimi representations, which they finds "very intuitive". In discussion with P4, the user correctly distinguished important nuanced differences between the `Parameter` and `Variable` composite data types because they match their intuition about what those terms mean with the environmental and economic modeling domains.

P8 started a model from scratch, modeling a commonly used physics simulation. After viewing the documentation and examples for a few minutes, they methodically flipped between the documentation and their evolving code, quickly and correctly writing out first the `Parameters` and then the `Variables` within discretized `Components`. Similarly, P4 and P1 were able to easily map the addition they wanted to make to an additional model to the concepts they needed to augment based on a scan of existing code.

Implication. Prioritizing closeness of mapping between domain specific concepts and embedded DSL constructs results in efficient use of an embedded DSL and helps users pick up and understand the language.

5.4 Personal engagement with community is crucial to attracting and retaining users.

The Mimi language was designed to support collaborative work by researchers, which led to elements like a modular design, use of an open-source host language, a well-defined and simple API for model modification and augmentation. We find that users highly value the existence of personal engagement with the Mimi community, both for (1) ease of collaboration and contribution and (2) receiving assistance.

A majority of our users started using Mimi because their research teams also used it, or because they saw the modular focus of the language as an opportunity to contribute to and work with other targeted groups or individuals. P1 said "seeing other papers using it and having the thought that at some point I could merge these components [with this work of other researchers]" was a large motivation for using Mimi. They saw it as "a way to get close to research groups" and that the "workshop sold it well", referring to a workshop carried out with 50 domain experts. As further evidence of this value, a majority of users were working on tasks to augment or modify existing models in Mimi as opposed to starting a brand new model or project from scratch.

Of the ability to contact developers with questions, P2 stated that "It has been absolutely crucial ... I could lose days on this and instead I just write you guys and in a few hours I have an answer it's incredible my friends are jealous". Similarly, when asked about their primary pain points P1 said that we could find a record of them in their posts on the (Mimi forum), where they turned when documentation was not enough. In fact, as opposed to their experience just searching for answers on Stack Overflow but never writing their own question due to the assumption that "it's gong to take years" to get a response, 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".

Head (2016) emphasizes the importance of a users perceived ability to get up-to-date, correct information and answers about a package from the community as key to adoption [20]. Meyerovich and Rabkin (2020) find that extrinsic social language features, including use in a group, team familiarity, and existing code to be among the top reasons for adoption

of general-purpose languages, even when compared directly to intrinsic language features [26]. Our study indicates we can apply these same lessons to adoption of domain-specific languages, in addition to general-purpose languages.

Implication. As pointed out directly by Meyerovich and Rabkin, in many cases "developer preferences here are shaped by factors extrinsic to the language". While decisions about intrinsic features of an embedded DSL is obviously important, designers will see significant benefits in adoption and retention if they pay attention to designing the systems to support community use and rapid, personalized assistance to users.

5.5 Users are sensitive to the time cost and error proneness of data I/O between different languages and file formats.

Modeling in the current "Big Data" era nearly always requires the input, processing, and output of large amounts of tabular data. We find that this is a significant pain point for users, and steps taken by an embedded DSL to minimize the manual nature and frequency of such processes is highly valued.

The session with P3 was largely taken up by the user creating and testing small data I/O functions to read in and format their data. This process was iterative, including viewing files within the IDE and running snippets of code to check outputs in the REPL. Another user, P6, was frustrated by trying to view a large tabular file in their IDE and as they scrolled through the CSV file stated "those are lots of things that are hard to look at there so I won't bother" and closed the file.

One design goal of Mimi is to represent 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 facilitate the linkages and dictate the flow of data from one component to the next. This structure enables users to build permutations of components from different existing models, or swap in components of their own and link together the data flow, *without ever needing to write data out to files and back in.* Previously, models were written in differing languages and platforms and not cleanly modularized, such that the same task may have required running one model in Fortran, extracting output, and inputting that data to a different model in Python. Both P2 and P7 explicitly describe the data I/O advantage of this approach. We also note that a majority of users take advantage of this feature, and we did not observe any need for I/O when running a model from start to finish (only upon initialization or output of final results).

P7 said of components from different models "it was cool to see them working together ... that you could add components from one model to another model was satisfying ... pre-Mimi it involved exporting CSVs of outputs form one model and importing it to another which is really slow and a bit clunky so it is nice ... it will pull inputs in directly rather than needing to export CSV of outputs and pull it back in."

Similarly, P2 stated "A top priority is to avoid language switching, especially in this work where doing so includes a lot of data transfer. Going back and forth between languages ... my top priority is trying to avoid doing that ... it's just too time consuming ... manipulating datasets from one thing to another ... I'm going to make mistakes ... it's similar to being stuck in Excel you make mistakes and you don't see them ... I prefer keeping everything within one language."

Implication. An embedded DSL intended to perform domain specific tasks involving data, and data I/O, can help users avoid tedious work and error proneness by prioritizing functionality that keeps data I/O simple and internal.

6 LIMITATIONS AND NEXT STEPS

A few limitations of this study are:

- A pool of nine participants is too small to support statistical analyses.
- The sample of participants may not be a representative sample, as we conducted recruitment mostly through personal channels—although we note we did post publicly to the Mimi forum and several braoder email lists and Slack channels.

Immediate next steps will include iterating, refining, revising, and augmenting the our findings. We also hope to add a thematic analysis of the textual data from the questions and answers on the forum, which should provide rich new information to validate our current findings, and introduce new ones.

7 CONCLUSIONS

This empirical study uses an embedded DSL in the climate change economics domain to provide insights on the usability of embedded DSLs, and how designers can write eDSLs with these lessons in mind. The contextual inquiry approach, combined with deductive thematic analysis, puts the focus of our analysis on observed participant behaviors. The primary findings of this study are grouped into five themes, each with a succinct implication for embedded DSL designers, and span both extrinsic and intrinsic qualities of these languages. In the future, we will consider adding a second dimension to this work by performing a thematic analysis of available textual data from an available public forum to enhance the current work.

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, Prof. Koushik Sen, the PLAIT lab at UC Berkeley.

REFERENCES

[1] 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.

[2] 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.

[3] 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*. 16–18.

[4] 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.

[5] 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.

[6] Ankica Barisic, Pedro Monteiro, Vasco Amaral, Miguel Goulão, and Monteiro Miguel. 2012. Patterns for evaluating usability of domain-specific languages. In *Proceedings of the 19th conference on pattern languages of programs (PLoP)*.

[7] 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.

[8] 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).

[9] 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.

[10] 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).

[11] 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.

[12] Steven Clarke and Curtis Becker. 2003. Using the cognitive dimensions framework to evaluate the usability of a class library. In *Proceedings of the First Joint Conference of EASE PPIG (PPIG 15)*.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] Paul Hudak. 1996. Building domain-specific embedded languages. *Acm computing surveys (csur)* 28, 4es (1996), 196–es.

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

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

[25] 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.

[26] 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.

[27] Brad A Myers, Andrew J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.

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

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

[30] 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.

[31] 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.

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

[33] 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.

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