Tales from 1002 Repositories: Development and Evolution of Xtext-based DSLs on GitHub

Weixing Zhang [1], Daniel Strüber [1,2]
[1]*Chalmers | University of Gothenburg, Gothenburg, SE* [2]*Radboud University, Nijmegen, NL*
{weixing,danstru}@chalmers.se

***Abstract*—Domain-specific languages (DSLs) play a crucial role in facilitating a wide range of software development activities in the context of model-driven engineering (MDE). However, there exists a significant gap in the systematic understanding of how DSLs evolve over time, which could hamper the development of effective methodologies and tools. To address this gap, we performed a comprehensive investigation into the development and evolution of textual DSLs created with Xtext, a particularly widely used language workbench in the MDE community. Through a systematic analysis of 1002 GitHub repositories, we explore DSL development practices with an emphasis on the involved artifact types, development scenarios, evolution activities, and the co-evolution of related artifacts. We find that the majority of analyzed languages were developed following a grammar-driven approach, although a notable number adopt a metamodel-driven approach. Additionally, we identify a trend of retrofitting existing languages in Xtext, illustrating the framework's flexibility beyond the creation of new DSLs. Addressing a need for large and systematically documented datasets in the model-driven engineering community, we contribute a dataset of repositories together with our collected meta-information, which can be used to inform the development of improved tools for supporting the development and evolution of DSLs.**

***Index Terms*—Xtext, software evolution, DSLs**

I. Introduction

Domain-specific languages (DSLs, [1]) are custom-tailored software languages addressing a particular domain of expertise. By providing a tool to create models on a suitable abstraction level, DSLs play an important role in model-driven engineering (MDE, [2]), where models created using a DSL can be used for a large variety of activities such as design, analysis, code generation, and testing.

Developing a DSL is a high-stakes activity. Previous design decisions often cannot be changed without significant effort on the part of the language developers and users. Still, a need to change the language may arise especially in the context of *language evolution*, where the developers add new features or respond to experience with the language [3]. In consequence, there is a need for sound methods, practices, and techniques for supporting the evolution of DSLs. However, to date, the development of support for DSL evolution is typically driven by the opinion of experts and individual cases encountered in their own practice or experience reports—Thanhofer et al. [4] provide a survey with 14 individual cases. Developers of future evolution methods would benefit from systematic knowledge about DSL evolution obtained from a larger number of cases.

To understand how MDE artifacts are developed and evolved, there is a trend towards large-scale studies that systematically collect evidence from open-source software (OSS) projects [5], [6], [7]. However, for development of DSLs in the context of MDE, such a study is not available yet.

In this paper, towards closing this gap, we contribute the first large-scale multiple-case study of DSL development and evolution. Based on a repository mining methodology [8], we collected and analysed data from 1002 GitHub repositories, to answer questions about involved artifact types, development scenarios, and evolution activities. Our study is focused on textual DSLs, specifically those developed using the Xtext framework [9], which is particularly widely used in the MDE community, due to its rooting in the Eclipse ecosystem. Xtext also serves as a blueprint for increasingly widely used DSL workbenches such as *textX* (https://pypi.org/project/textX/) and *langium* (https://langium.org/).

As part of our contribution, we provide a dataset [10] of 1002 repositories (via their URLs) together with our extracted meta-data, e.g., the repository's type, employed development scenario, availability of various artifacts, and change statistics. This dataset addresses need for large and consistently documented artifacts expressed in the MDE community [11], [12] and can be particularly useful for follow-up research, both to develop advanced (e.g., AI-based) techniques, as well as supporting the identification of cases that can be used to inform design and evaluation activities.

Specifically, we focus on the following research questions:

RQ1: *Are there GitHub projects that use Xtext? Which are these projects?* We set out to investigate basic information of Xtext-related activity in GitHub repositories. Our contribution includes a manual classification of all repositories, highlighting 228 repositories that include a worked-out and well-documented language, which are particularly useful for follow-up research such as our RQ2 and RQ3. We further characterize the repositories based on application domains and Xtext artifacts, which can aid the selection of comprehensive examples for research and tool development purposes.

RQ2: *Which development scenarios for Xtext-based languages are applied in these projects?* Xtext supports multiple development scenarios that differ in their complexity (discussed later). There is a question on whether complex development scenarios such as meta-model-driven development are used in practice and thus, need to be supported with dedicated approaches. Moreover, in the course of answering this question, we discovered a trend of what we call *retrofitting*—creating an Xtext grammar that fits an existing language.

RQ3: *How do Xtext-based langages on GitHub evolve over time?* Since DSLs are often envisioned as "small" languages [13], it is tempting to view their evolution as a non-issue. In this RQ, we study longitudinal aspects of language projects, including their longevity and amount of changes performed. We highlight the existence of long-living language projects and shed light on their proneness to significant changes, leading to challenges that we discuss later, in Sect. VI of the paper.

II. Background

Xtext. The development of DSLs is supported by a large variety of existing workbenches [14]. In this paper, we focus on the Xtext workbench [9], because we are interested in language development in MDE contexts, which provides various benefits for language development. First, the possibility to apply a rich ecosystem of existing MDE tools and techniques to the developed languages and their models. Second, support for blended modeling [15], a rising paradigm in which several different concrete syntaxes (e.g., graphical and textual) are provided for the same underlying meta-model, which allows the developers to choose one that fits best for the task at hand. In an MDE context, Xtext is the most widely used technology for textual DSL development, and naturally supports blended modeling, since it involves the use of meta-models for abstract syntax specification, and goes beyond meta-modeling by also allowing to define a textual concrete syntax.

Xtext allows the specification of a textual DSL in terms of an extended EBNF grammar, where the extensions are mappings of language elements to an underlying meta-model (in EMF [16]). The metamodel specifies the language's abstract syntax (language concepts and their relations), whereas the grammar specifies the concrete syntax (keywords, parentheses, nesting of elements) with the mapping to the abstract syntax. From the provided specification, Xtext can automatically generate comprehensive tool support, including a textual editor with automated checks, syntax highlighting, and auto-formatting. While Xtext is rooted in the Eclipse ecosystem, adapters for other IDEs (e.g., IntelliJ) are available.

In our repository mining context, we identify grammars and meta-models, as well as two additional artifact types, as distinct file types. In Xtext-based DSL development, there are the following four main MDE artifacts: 1) Xtext grammar files, 2) Ecore metamodel files, 3) modeling workflow engine (MWE) files, and 4) textual instance files. MWE files support the orchestration of automated activities, in particular, the generation of modeling components. Among others, they define the file extension for textual instances (a.k.a. models created using the language), which render them interesting for our study of artifacts. Xtext files can be generated from Ecore files, and conversely, Ecore files can also be generated from Xtext files [9]. An MWE file is used to generate Xtext artifacts from Xtext grammar [9], and these Xtext artifacts are used to generate a textual editor for the DSL. Textual instance files are edited in this textual editor.

Development Workflows and Scenarios. To specify a DSL's concrete and abstract syntax, Xtext uses two separate artifacts–a grammar and meta-models—, which leads to the challenge of keeping them synchronized with each other as the language evolves. To this end, Xtext supports two main development workflows [9]: In a *grammar-driven* scenario, the user primarily edits the grammar, and has changes propagated to the meta-model by completely re-generating it. In a *metamodel-driven* scenario, the user primarily edits the meta-model. After the changes to the meta-model, the grammar has to be updated, which in the default process has to be done manually–re-generating the grammar is not feasible due to potential information loss about the concrete syntax.

Choosing an appropriate workflow requires to consider the context in which the language is developed. The meta-model driven workflow is useful in scenarios where the meta-model has to be manually managed, e.g., when it is the center of an already-existing ecosystem of tools, when it comes from a third party vendor or standardization committee, or in a blended modeling scenario with several concrete syntaxes. The grammar-driven workflow is simpler to support and, therefore, generally preferable in other scenarios.

III. Related Work

DSL Evolution. The evolution of DSLs has been studied so far with a focus on providing improved evolution support, and on reporting individual cases of evolving DSLs. Both are studied in a systematic mapping study of Thanhofer-Pilisch et al. [4], the results of which can help researchers and practitioners working on DSL-based approaches obtain an overview of existing research on and open challenges of DSL evolution. We now discuss selected cases with a particular focus on industrial experiences. Mengerink et al. investigated the evolution of DSLs in a large industrial MDSE ecosystem [17], through which they summarized common evolution types and evaluated the automation capabilities of evolution in real-life scenarios. Schuts et al. reported in [18] their experience in evolving a Philips-owned DSL, with the goal of enabling the DSL to support a range of Philips systems. This is concrete experience from the industry regarding the evolution of DSL. DSL evolution generally leads to issues with keeping multiple involved artifacts synchronized with each other [3]. In the MDE community, a plethora of work on co-evolution problems has evolved, in particular metamodel-model co-evolution [19], [20], [21], where changes to the metamodel make evolution of the associated models and transformations necessary. As we discuss later, our dataset and results could inform the development of new approaches in this area.

Mining. Due data availability and volume, GitHub has become the data source of choice for repository mining research [22], including those in model-driven engineering. In [6], Shrestha et al. mined MATLAB/Simulink-related repositories and assembled a large corpus of Simulink projects, which includes model and project changes and allows redistribution. Mengerink et al. used software repository mining to create a large corpus of OCL constraints [23]. Previous studies of EMF metamodels focused on collecting EMF models from Eclipse projects [24], studying the use of meta-modeling concepts in GitHub projects [7], and on deriving a high-quality dataset for machine learning [25]. Hebig et al. [5] investigated the use of UML in OSS projects by systematically mining GitHub projects. No previous study focused on Xtext-based DSLs in GitHub repositories.

The only previous work that explicitly applied repository mining to Xtext grammars (among other MDE artifacts) is MAR, a search engine for models [26]. MAR offers a by-example query mechanism for searching a database of 600K models retrieved from existing repositories. While their underlying dataset includes Xtext models from GitHub, it is not annotated with the metadata offered in our dataset (e.g., number of instances, development scenario, evolution statistics). Moreover, our research contribution and questions have a different scope, focused on characterizing Xtext-specific projects, with their development and evolution scenarios.

IV. Methodology

We now describe our used repository mining methodology [22]. Like the previous work discussed above, our study was focused on GitHub, the largest existing software repository platform. Our overall process is divided into 6 steps. First, we obtained a list of non-fork open-source repositories containing Xtext files from GitHub. Second, we cloned all the retrieved open-source repositories to a local hard drive to facilitate access and information acquisition. Third, we searched for relevant file types in these repositories by file extensions and collected information about them. Fourth, we analyzed the development scenario of each repository. Fifth, using the repository-specific file extensions of the instances that we identified from MWE files, we collected the instances of these extensions and calculated information about them. Sixth, we manually classified all obtained repositories, before analyzing the collected data with respect to our research questions.

Step 1: Data Collection. We used the GitHub API to obtain repositories that are related to Xtext. Since the most fundamental MDE artifact of an Xtext project is its grammar, which is stored as a file with the extension *.xtext*, our search string was based on the main clause `q?=.xtext` – that is, we searched repositories that contain a file with that extension. We later observed that this search string partially lead to the identification of repositories that did not actually contain an Xtext file, but were still related to Xtext in a different way, as described below. Furthermore, to exclude repositories that are forks of other repositories, as these might mostly replicate the information from the original repositories and thus bias our results, we set the parameter "fork" to "false".

One complication was that the GitHub API only allows access up to 1,000 results, even when using the *pagination* feature. However, from a trial search on the GitHub website, we observed that the number of relevant repositories may exceed 1,000. Hence, added a third parameter to the request, which is the creation time of the repository. We use January 1, 2018, as the boundary to divide the request into two, i.e., requesting results before that date and from that date. We retrieved six pages with 576 repositories created before January 1, 2018, and five pages with 426 repositories created from this date, leading to a total of 1002 repositories.

We developed a Python script to complete the above work. Its functions included setting request parameters, sending requests, and dumping request results into local text files. Execution of this script only took a few seconds to complete. The rationale for developing a new script, instead of starting from an existing dataset (e.g., [26]) was that it allowed us to retrieve the most up-to-date information from GitHub and that it naturally integrated with our remaining analysis activities. The overall query used for the requests had the following form:

```
https://api.github.com/search/repositories?
q=.xtext+created:{since_date}..{stop_date}
+fork:false&page={page}&per_page=100
```

The repository information we obtained contains various information about the repository, such as the repository's ID, name, whether it is private, owner, html_url, and description. We developed another script to extract the name, owner's login, and html_url of these repositories from the text files and store them in a table to facilitate subsequent data mining and analysis. The result was a table containing 1002 rows.

Step 2: Repository Cloning. GitHub allows to obtain information about GitHub repositories through the GitHub API. However, GitHub has restrictions on the rate and frequency of access. Since we in subsequent analysis needed to frequently access different and large numbers of files, we decided to clone all repositories to a local hard drive and further analyze the local clones. We only cloned the `master` branch (except for a few cases where `master` was empty, where we manually identified a different main branch instead), leaving an analysis of use of branching and pull request as future work.

Step 3: File Search. Given the local clones of the 1002 identified repositories, in Step 3, we identified their contained grammars, meta-models, and workflow files, by searching for files with the extensions ".xtext", ".ecore", ".mwe2", respectively. We recorded both the count of files, as well as additional information. e.g., the directories in which files were found, which we used in subsequent steps.

Step 4: Scenario Judgement. In Step 4, we determined which the used language development scenarios in each repository. Xtext supports two language development scenarios, described in Sect. II. In the grammar-driven scenario, the text definition of the Xtext grammar has a statement starting with the keyword "generate", which results in generating a meta-model from the grammar. When generating the metamodel, Xtext automatically places the metamodel in a folder named "generated". In the metamodel-driven scenario, language developers create a metamodel in a folder they create. We assumed that the developers do not name their created folders "generated", which would be counterintuitive. Considering that there may be multiple Ecore files in a repository, we distinguished three cases in which Ecore files existed in a repository: 1) All Ecore files in the repository are in a folder named "generated", 2) all the folders containing Ecore files in the repository are not named “generated", and 3) some of the folders containing Ecore files in the repository are named “generated" and some are not. We classify the first situation as a grammar-driven scenario, the second situation as a meta-model-driven scenario, and the third situation refers to both scenarios.

Step 5: Instance Search. One Xtext artifact type we set out to study was instances (models); yet, identifying instances is non-trivial, as their file extension differs per language. We identified instances by reading the file extension from the previously identified MWE files and then searching relevant files. To check whether the found instances adhere to the grammar in the same repository, we performed a sampling analysis: we randomly selected ten repositories that contained both grammar and instances and manually checked conformance.

Step 6: Classification. To give deeper insights into the different kinds of Xtext-related repositories on GitHub, in Step 6, we manually classified repositories into different types and analyzed the frequency of different types. Our process for this was as follows: First, one author manually labeled a sample of 200 repositories with improvised labels. The labels emerged from the observations that a few specialized categories were recurring between the repositories, with proper language projects (described below) being of most interest for our study. Second, in a discussion between the authors, the obtained labels were harmonized, by defining descriptions and explicit criteria for them. Third, we labeled the complete set of all repositories with the final set of labels. The final labeling was done by one author and checked by another author. Disagreements were resolved together.

The obtained list of types together with their descriptions and criteria was as follows.
Language: A repository with proper, documented Xtext-based language. *Criterion:* The README.md or “About” section describes it as (implementation of) a language, or a software system that incorporates a clearly identifiable language. *Notes*: We also collected the language’s domain. After noticing that several repositories re-implemented an existing language (a phenomenon we call *retrofitting*), we also noted whether this is a case for each repository.
Training/Examples: A repository serving the training of Xtext users, usually in the form of an example, tutorial or both. *Criterion:* The project’s README.md or “About” section describes it as an example, a tutorial, or a demonstration. *Note*: using the word "example" as part of the name was not deemed as a useful criterion, as examples might be created for experimental purposes—see below.
Infrastructure: A repository with tooling for supporting development with Xtext. *Criterion:* The README.md or “About” section suggests that the project is about supporting tooling.
Experimental/Personal: A repository that does not fall in any of the above categories, but is still directly related to the language workbench Xtext. *Criteria:* Any of the following applies: 1. The contained grammar is extremely small and basic. 2. The contained grammar is taken from a standard example provided with Xtext. 3. The README.md and “About” section are empty or give no context information on what the repository is about. 4. The README.md and “About” section describes it as an assignment submission for a course, or as an example for debugging purposes.
Unrelated: A repository unrelated to the language workbench Xtext, except for naming. *Criterion:* The only connection to the language workbench is sharing (parts of) the name.

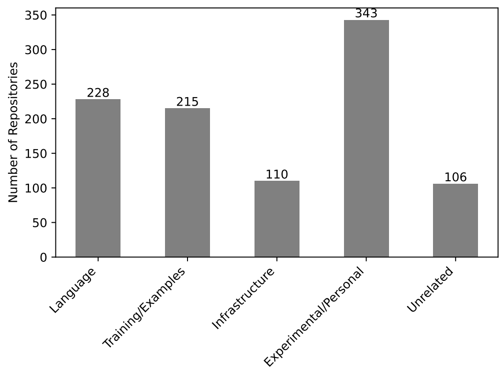

Fig. 1. Classification of repositories.

V. Results

This section presents the results of our investigation. In this research, an ample amount of data is collected and analyzed, mostly in an automated way, using scripts developed by the authors. The resulting dataset (spreadsheet) and our analysis scripts are available from the associated artifact [10].

RQ1: Xtext in GitHub Repositories. To overview the 1002 repositories identified via our search methodology, we show the outcome of our manual classification, according to the methodology explained in Section IV. As indicated in Figure 1, we found 228 repositories in which languages have been developed and contain descriptions of them. There are 215 repositories with documented training and example materials. 110 repositories provide infrastructure to support development with Xtext. Repositories classified as *experimental/personal* are the most numerous, with 343 cases. Additionally, 106 repositories have no relationship to Xtext except for naming.

We illustrate the different categories with examples. *Languages* developed with Xtext span a large variety of cases, from DSLs for specific application domains such as physiotherapy exercises (`Kinect-ECE-XText`), telemedicine (`telemed`), document management (`Xarchive`), to technical domains such as JSON schema (`xtext-json`), Quantum computing (`Quingo/compiler_xtext`) and Eclipse launch configurations (`lcdsl`). A noteworthy sub-category, discussed in RQ2, are cases of retrofitting existing languages, such as GraphQL (`graphQL-xtext-grammar`), or Oberon (`Oberon-XText`). *Training/Example* projects comprise tutorials such as 15-minute Xtext tutorial in Chinese (`xtext_tutorial_15_min_zh`). *Infrastructure* projects include technology for integrating Xtext and particular languages in specific contexts, e.g, build processes (`gradle-xtext-generator`) and editors (`vim-xtext`). *Experimental/personal* code often involves a dump of the

TABLE I
File type frequency in projects containing Xtext files.

	Xtext		Ecore		MWE	
#Files	#Repo	Perc.	#Repo	Perc.	#Repo	Perc.
>= 100	6	0.83%	6	0.83%	1	0.14%
10 - 99	19	2.63%	8	1.11%	19	2.63%
2 - 9	261	36.15%	108	14.96%	264	36.57%
1	436	60.39%	352	48.75%	410	56.79%
0	/	/	248	34.35%	28	3.88%

user's personal workspace (e.g. `Xtext Workspace`). *Unrelated* repositories generally result from a name clash, such as using the name 'xtext' for some unrelated tool (e.g., `resloved`'s text displaying tool `xtext`), or within some longer name, such as `TopXTextUI`.

With Xtext grammars being one of the core artifacts of Xtext projects, we checked how many are contained in each repository. The results, shown in Table I, show that of these 1002 repositories, only 722 really contain at least one xtext file. Unsurprisingly, nearly all repositories classified as *language* contained an Xtext grammar—219 out of 228 cases. The nine exceptions involved repositories that contained Xtext-based editors for a particular language without providing the underlying grammar, such as `Palladio-Editors-VSCode`, and ports of originally Xtext-based DSLs to other workbenches, such as `eJSL-MPS` [27].

For the 722 repositories that contain at least one Xtext grammar file, Table I reports the numbers of other included MDE artifacts. Of these repositories, 248 contain no Ecore metamodel file, while 352 repositories contain one Ecore metamodel file. A total of 6 repositories contain at least 100 Xtext and Ecore files each. All of these are infrastructure projects that use a larger number of examples for testing and/or demonstration purposes; three of them associated with the official Xtext project. The relatively large number of projects that contain an MWE file but no Ecore file can be explained by the fact that in a grammar-driven scenario, Ecore files can be fully automatically generated from the underlying grammar, and it is a common practice to not commit automatically generated artifacts to repositories.

Table II shows the results of identifying textual instances in the 722 repositories that contain at least one Xtext grammar file, 446 contain no textual instances at all, 103 contain only one textual instance, and 173 repositories contain at least two textual instances. Interestingly, we found two repositories containing more than 1000 textual instances, namely, `xtext/xtext-monorepo` and `eclipse/xtext`, owned by the Xtext team and the Eclipse Foundation, respectively.

As useful meta-information to inform the identification of particularly widely used repositories, we collected the number of forks and stars for all repositories and included them in our dataset [10]. We found that 219 repositories had forks and 322 repositories had stars.

Results of RQ1: We find five categories of repositories—language, training/examples, infrastructure, experimental/personal and unrelated. A total of 722 repositories contain one or several xtext grammar files. About two-thirds of them contain at least one Ecore metamodel file.

RQ2: Development Scenarios. To answer RQ2, on development scenarios for language development, we specifically focused on those repositories classified as *language* and judged their development scenarios as described in Section IV, leading to the results shown in Table III.

There are a total of 228 repositories classified as *language*, the majority of them (i.e. 169) are in a grammar-driven scenario, and 66 of them have developed an Xtext version of an existing language. 41 of the 200 repositories are in a metamodel-driven scenario, and 6 of them have developed an Xtext version of an existing language. Nine of these 228 repositories contain both grammar-driven and metamodel-driven scenarios, and none of them have developed an Xtext grammar for any existing language. Additionally, the nine repositories that were classified *language* but did not contain Xtext files (described in the results for RQ1), were not suitable for our analysis and hence excluded from it.

We give selected examples for the identified metamodel-driven cases since it is arguably the more complex scenario, involving manual overhead for keeping meta-models and grammars synchronized. `megal-xtext` provides a textual syntax for the MegaL mega-modeling language, a language that by design provides several concrete syntaxes [28], a typical motivation for the metamodel-driven scenario. `Kotlin-Meta-Model` is a repository with the main claim of providing a meta-model for the Kotlin JVM language; the provided Xtext grammar is mentioned as an additional artifact. Other repositories such as `QuestionnaireDSL` do not include a definite explanation for following the metamodel-driven scenario, but at least contain a visualization for the metamodel (*aird* file), which indicates an intention to explicitly design the metamodel. Repositories classified as *both* generally comprise several languages with different workflows, e.g.,

TABLE II
Instances in repositories that contain Xtext grammars.

Count of Instances	#Repos	Percentage
>= 1000	2	0.28%
100 - 999	10	1.39%
10 - 99	31	4.29%
2 - 9	130	18.00%
1	103	14.27%
0	446	61.77%

TABLE III
Frequency of language development scenarios.

Scenario	#Repos	#Retrofitting
grammar-driven	169	66
metamodel-driven	41	6
both	9	0
not applicable	9	2

telemed with separate languages for information storage and querying of telemedicine information.

A noteworthy activity that our manual classification of repositories brought forward is *retrofitting*: the implementation of some existing language in Xtext. We found 74 cases of repositories that can be classified as such. In several cases, repositories included implementations of popular practical languages, e.g., GraphQL for graph database querying (graphQL-xtext-grammar), JSON Schema for schema definition (JSON-Schema-to-internal-DSL), and PlantUML for lightweight UML diagram creation (plantuml-eclipse-xtext). The added value of an Xtext implementation for these languages is to benefit from the editing support offered by Xtext, e.g., automated code completion, syntax highlighting, and formatting. A few cases were concerned with historical programming languages and could be seen as an enthusiastic effort (e.g., Oberon-IDE).

The majority of retrofitting cases was developed in a grammar-driven way, which is consequential: The concrete syntax in these cases is fixed and hence, it is natural to specify a grammar matching the existing syntax, thus retrofitting it.

Results of RQ2: While the majority of Xtext-based language development projects involve the grammar-driven scenario, a nonnegligible number relies on the metamodel-driven one. We identify *retrofitting*—creating an Xtext implementation of an existing language—as a noteworthy development activity.

RQ3: Evolution of MDE Artifacts in Xtext-Based Projects. To investigate how languages and their artifacts evolve, we focused again on the 228 repositories classified as 'language'.

We were interested in longevity of language projects and the activity around their included grammars. To study this aspect, we focused on those *language*-classified repositories that contained a grammar, leading to 219 repositories. We observed a time span for updates of the repository and the contained grammar. The time span is calculated as the difference in days between the first and the last commit. We obtained the results shown in Table IV. The results show that 43 repositories were never updated after they were created, while 36 repositories had updates spanning more than 1,000 days, meaning that they were still maintained after a substantial amount of time.

TABLE IV
TIMESPANS OF COMMITS IN REPOSITORIES AND XTEXT FILES.

	Repo		Xtext	
Timespan (Days)	#Repo	Perc.	#Repo	Perc.
>= 1000	36	16.43%	18	8.22%
100 - 999	55	25.11%	43	19.63%
10 - 99	59	26.94%	44	20.09%
1 - 9	26	11.87%	29	13.24%
0	43	19.63%	85	38.81%

TABLE V
AVERAGE CHANGE TO NUMBER OF LINES AND RULES WHEN COMPARING FIRST AND LAST COMMITED VERSION OF XTEXT GRAMMAR.

Avg. Change to #Lines	#Repos	Percentage
< 0	4	5.02%
0	78	35.62%
> 0 and <= 10	27	12.33%
> 10 and <= 100	68	31.05%
> 100 and <= 1000	33	15.07%
> 1000	2	0.91%
Avg. Change to #Rules	**#Repos**	**Percentage**
> 100	3	1.37%
<= 100 and > 10	46	21%
<= 10 and > 0	65	29.68%
= 0	91	41.55%
< 0	14	6.39%

Considering the time span of updates to the grammar included in the projects, there are many (i.e. 85) repositories where Xtext files are never updated after they are initially created. There is also a nonnegligible number (i.e. 18) of repositories that still have records of updating Xtext grammar files after a long time (no less than 1000 days).

We were further interested in how much Xtext grammars change over time. To this end, we considered both the changes to the overall number of lines and the number of rules. Starting with the number of lines, for those 219 repositories that contain at least one Xtext grammar file and are classified as *language*, we compared the first and the last committed version of the Xtext files in terms of their line counts, reporting the averages per repository in Table V. We can see that the number of lines of text for Xtext grammar has not changed in 35.62% of the repositories. For those repositories that have changed, the vast majority of them contain Xtext grammar files that have basically added the number of lines of text. Among them, the average increased line number of Xtext grammar files in 35 repositories by more than 100.

We also analysed the changes in the number of grammar rules in the Xtext grammar file, leading to the results shown in Table V. We can see that in more than 40% of the repositories, the number of grammar rules in the Xtext grammar has not changed. Part of the reason is that 66 repositories have not updated their Xtext files since they were initially committed. For those repositories where the number of rules of the Xtext grammar changed, in most of them the number of rules increased over time. In particular, the average number of added grammar rules in the Xtext grammar contained in 47 repositories exceeds ten.

Our dataset [10] contains additional metadata quantifying evolution activities, specifically, the change frequency of grammars, metamodels, and instances per project. While a detailed analysis is outside the scope of this paper, we observe the following trends: As one would expect, grammars are updated more often than meta-models. However, the difference in update frequency is more pronounced in the grammar-driven workflow than in the metamodel-driven one, which might suggest that users in this workflow spend more time polishing concrete syntax aspects. Instances are more likely to be updated in repositories with more Xtext grammar updates.

Results of RQ3: We find a spectrum of project and language development lifespans of Xtext projects on GitHub, with a nonnegible part (8%) of grammars still being updated 1000 days after the initial commit. The amount of change performed in individual languages can be significant, with more than 10 rules being added in 22% of all languages.

VI. Discussion

Need for approaches for co-evolution in textual DSL evolution contexts. Our analysis in RQ3 reveals the existence of DSL evolution projects that are maintained over several years and involve significant changes to the language over time. In such projects, challenges arise from the synchronized evolution of all involved artifacts, including grammars, meta-models, instances, and workflow files. In a general MDE context, synchronized evolution is a well-studied area, termed *co-evolution* or *coupled evolution*. A plethora of existing work [29], [30], [31], [32], [33], [34], [35] provides foundational approaches for managing co-evolution between meta-models and other artifacts, typically models and transformations. However, there remains a significant gap in tool support and methodologies specifically tailored to the co-evolution of artifacts involved in textual DSLs developed with frameworks like Xtext, with their focus on both meta-models and grammars. To address some of the complexity in the meta-model-driven scenario (RQ2), one could rely on principles from the grammarware sphere and support an operator-based approach to grammar evolution [36]. In our recent work, we follow up on this idea to automate parts of the synchronization of the grammar after meta-model changes [37], [38], [39], [40], [41]. Another open challenge is grammar-instance co-evolution, which could benefit from the available foundational approaches for metamodel-model co-evolution, but needs to deal with concrete syntax aspects of grammars. This problem has not been addressed yet in the technical space of Xtext. Lämmel's LAL approach [42] could be useful for validating a solution that addresses it.

Future research should focus on integrated approaches and tools that facilitate the simultaneous evolution of all related DSL artifacts. The comprehensive dataset compiled from our study, particularly the 196 cases where all crucial artifacts such as grammar, metamodel, and instances are available, presents a valuable resource for developing new co-evolution approaches, by supporting their design, testing and evaluation.

Follow-up studies investigating change types and scopes. Our comprehensive dataset paves the way for follow-up studies on language development and evolution, particularly by expanding on RQ2 and RQ3 to analyse the changes to projects at a finer level of detail. Future research should perform additional analysis of the specific patterns of evolution that languages undergo, identifying common trajectories and deviations in their lifecycle, which would expand on earlier studies focused on a single project or ecosystem, such as those discussed in Sect. III. Moreover, an under-explored area is the extent to which the features provided by frameworks like Xtext are utilized or underused within real-world projects. Like Babur et al.'s study [7] for the case of Ecore metamodels, such investigations can uncover valuable insights into the actual needs and practices of language engineers, leading to more targeted improvements in language workbenches. Additionally, examining the types of changes—whether they pertain to syntax, semantics, tooling, or documentation—can shed light on the multifaceted nature of DSL evolution and the challenges it poses. This deeper understanding can inform the development of better support tools and methodologies, ultimately fostering more robust and adaptable languages.

Threats to validity. Threats to internal validity arise from us relying on the GitHub API. Since we cannot access the implementation of the GitHub API, we cannot verify *completeness*: the implementation could be inexact, which could lead to repositories not being captured by our query. As a safeguard, we checked whether a total of three expected repositories personally known to us appeared in our results, which was the case. Nevertheless, anecdotally, a colleague to whom we made available our dataset informed us about a project that was not part of it. Still, our findings that arise from a substantial number of cases and highlight the existence of understudied phenomena—meta-model-based evolution, retrofitting, long-living Xtext projects–do not require completeness to be valid.

Furthermore, repositories can be duplicates of each other, which might bias the results. For our *language*-classified repositories, which we investigate in RQ2 and RQ3, we checked that this is not the case and can exclude duplicates. For the *experimental/personal* category, which generally does not make any statements about the quality about the included repositories, anecdotally, some repositories have the same contents as others, potentially arising from having followed the same tutorial with the basic Xtext examples.

Considering external validity, our scope is restricted to GitHub and Xtext, both being particularly popular and widely used technologies in their respective communities. There is a larger variety of existing language workbenches [14], not all of which might equally benefit from our findings. The results from our study could be transferable to other workbenches that use a similar strategy for separating abstract and concrete syntax specification like Xtext, such as textX and langium. Yet, transferring our results to language workbenches that employ an entirely different paradigm (e.g., in the case of MPS, projectional editing) might be infeasible.

Considering construct validity, to mitigate the impact of subjectivity on our classification, we extensively discussed the criteria and problematic cases and eventually found consensus for all of them. Our classification further relies on documentation provided by the repository owners, which might not always be accurate or complete, leading two consequences: First, we did not investigate whether repositories were from industry or academia, which generally was not possible to tell from the documentation. Second, some of our *language*-classified repositories, in particular, among those classified as *retrofitting*, might be exclusively intended for self-teaching or demonstration purposes, but this context is unavailable to us. Users of our dataset are advised to use it in a way that makes sure that their assumptions are met, for example, taking into account our collected change history meta-information to identify cases with a rich evolution history.

VII. Conclusion

In our analysis of 1002 GitHub repositories, we found that the majority of our considered languages are developed following a grammar-driven approach, albeit a notable number adopt a metamodel-driven approach. We observed long-running projects with numerous changes, highlighting the diversity in DSL development and the need for tools and methodologies that can accommodate different scenarios. The trend of retrofitting existing languages in Xtext showcases its flexibility beyond creating new DSLs. Our dataset supports further research into DSL evolution and the development of methods to facilitate this evolution.

Future work will investigate our dataset further to uncover more insights and study detailed evolution patterns, particularly in Xtext constructs and code artifacts. We also aim to develop better methods for synchronizing artifacts in evolution scenarios, especially co-evolution between a grammar and its instances, a gap not addressed by current solutions.

References

[1] T. Kosar, S. Bohra, and M. Mernik, “Domain-specific languages: A systematic mapping study,” *IST*, vol. 71, pp. 77–91, 2016.

[2] T. Stahl and M. Völter, *Model-driven software development: technology, engineering, management*, 2006.

[3] R. Lämmel, *Software Languages*, 2018.

[4] J. Thanhofer-Pilisch, A. Lang, M. Vierhauser, and R. Rabiser, “A systematic mapping study on DSL evolution,” in *SEAA*, 2017, pp. 149–156.

[5] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, “The quest for open source projects that use UML: mining GitHub,” in *MODELS*, 2016, p. 173–183.

[6] S. L. Shrestha, A. Boll, S. A. Chowdhury, T. Kehrer, and C. Csallner, “Evosl: a large open-source corpus of changes in simulink models & projects,” in *MODELS*, 2023, pp. 273–284.

[7] Ö. Babur, E. Constantinou, and A. Serebrenik, “Language usage analysis for EMF metamodels on GitHub,” *Empirical Software Engineering*, vol. 29, no. 1, p. 23, 2024.

[8] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The promises and perils of mining GitHub,” in *MSR*, 2014, pp. 92–101.

[9] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*, 2016.

[10] W. Zhang and D. Strüber, “Dataset for ’Tales from 1002 Repositories: Development and Evolution of Xtext-based DSLs in GitHub Projects’,” 2024. [Online]. Available: https://osf.io/348qg/?view_only=94e940c6359d4374badd81374358f4fd

[11] G. Robles, M. R. Chaudron, R. Jolak, and R. Hebig, “A reflection on the impact of model mining from GitHub,” *IST*, vol. 164, p. 107317, 2023.

[12] C. D. N. Damasceno and D. Strüber, “Quality guidelines for research artifacts in model-driven engineering,” in *MODELS*, 2021, pp. 285–296.

[13] A. V. Deursen and P. Klint, “Little languages: little maintenance?” *JSEP*, vol. 10, no. 2, pp. 75–92, 1998.

[14] S. Erdweg, T. Van Der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, “Evaluating and comparing language workbenches: Existing results and benchmarks for the future,” *COMLAN*, vol. 44, pp. 24–47, 2015.

[15] I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J.-P. Steghöfer, and R. Hebig, “Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study,” *SoSyM*, vol. 22, no. 1, pp. 415–447, 2023.

[16] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*, 2008.

[17] J. G. Mengerink, B. van der Sanden, B. C. Cappers, A. Serebrenik, R. R. Schiffelers, and M. G. van den Brand, “Exploring DSL evolutionary patterns in practice: a study of DSL evolution in a large-scale industrial DSL repository,” in *MODELSWARD*, 2018, pp. 446–453.

[18] M. Schuts, M. Alonso, and J. Hooman, “Industrial experiences with the evolution of a DSL,” in *DSM*, 2021, pp. 21–30.

[19] G. Wachsmuth, “Metamodel adaptation and model co-adaptation,” in *ECOOP*, 2007, pp. 600–624.

[20] M. Herrmannsdoerfer, S. Benz, and E. Juergens, “COPE-automating coupled evolution of metamodels and models,” in *ECOOP*, 2009, pp. 52–76.

[21] R. Hebig, D. E. Khelladi, and R. Bendraou, “Approaches to co-evolution of metamodels and models: A survey,” *IEEE TSE*, vol. 43, no. 5, pp. 396–414, 2016.

[22] G. Gousios and D. Spinellis, “Mining software engineering data from GitHub,” in *ICSE Companion*, 2017, pp. 501–502.

[23] J. G. Mengerink, J. Noten, and A. Serebrenik, “Empowering OCL research: a large-scale corpus of open-source data from github,” *Empirical Software Engineering*, vol. 24, pp. 1574–1609, 2019.

[24] S. Kögel and M. Tichy, “A dataset of EMF models from Eclipse projects,” 2018.

[25] J. A. H. López, J. L. Cánovas Izquierdo, and J. S. Cuadrado, “ModelSet: a dataset for machine learning in model-driven engineering,” *SoSyM*, pp. 1–20, 2022.

[26] J. A. H. López and J. S. Cuadrado, “An efficient and scalable search engine for models,” *SoSyM*, vol. 21, no. 5, pp. 1715–1737, 2022.

[27] D. Priefer, W. Rost, D. Strüber, G. Taentzer, and P. Kneisel, “Applying MDD in the content management system domain,” *SoSyM*, vol. 20, no. 6, pp. 1919–1943, 2021.

[28] J.-M. Favre, R. Lämmel, and A. Varanovich, “Modeling the linguistic architecture of software products,” in *MODELS*, 2012, pp. 151–167.

[29] J. García, O. Diaz, and M. Azanza, “Model transformation co-evolution: A semi-automatic approach,” in *SLE*, 2012, pp. 144–163.

[30] D. E. Khelladi, H. H. Rodriguez, R. Kretschmer, and A. Egyed, “An exploratory experiment on metamodel-transformation co-evolution,” in *APSEC*, 2017, pp. 576–581.

[31] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, W. Schwinger, and J. Schönböck, “Consistent co-evolution of models and transformations,” in *MODELS*, 2015, pp. 116–125.

[32] S. Vaupel, D. Strüber, F. Rieger, and G. Taentzer, “Agile bottom-up development of domain-specific IDEs for model-driven development,” in *FlexMDE*, 2015, pp. 12–21.

[33] W. Kessentini and V. Alizadeh, “Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search,” in *MODELS*, 2020, pp. 68–78.

[34] D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, “A modeling assistant to manage technical debt in coupled evolution,” *IST*, vol. 156, p. 107146, 2023.

[35] D. Di Ruscio, L. Iovino, and A. Pierantonio, “A methodological approach for the coupled evolution of metamodels and ATL transformations,” in *ICMT*, 2013, pp. 60–75.

[36] V. Zaytsev *et al.*, “Negotiated grammar evolution,” *JOT*, vol. 13, no. 3, pp. 1–22, 2014.

[37] W. Zhang, R. Hebig, D. Strüber, and J.-P. Steghöfer, “Automated extraction of grammar optimization rule configurations for metamodel-grammar co-evolution,” in *SLE*, 2023, pp. 84–96.

[38] W. Zhang, J. Holtmann, D. Strüber, R. Hebig, and J.-P. Steghöfer, “Supporting meta-model-based language evolution and rapid prototyping with automated grammar transformation,” *JSS*, vol. 214, 2024.

[39] W. Zhang, J.-P. Steghöfer, R. Hebig, and D. Strüber, “A rapid prototyping language workbench for textual DSLs based on Xtext: Vision and progress,” *arXiv preprint arXiv:2309.04347*, 2023.

[40] W. Zhang, “Towards automated support for the co-evolution of meta-models and grammars,” Licentiate Thesis, University of Gothenburg, Sweden, 2023.

[41] W. Zhang, R. Hebig, J.-P. Steghöfer, and J. Holtmann, “Creating python-style domain specific languages: A semi-automated approach and intermediate results.” in *MODELSWARD*, 2023, pp. 210–217.

[42] R. Lämmel, “Coupled software transformations—revisited,” in *SLE*, 2016, pp. 239–252.