Mining the Usage of Reactive Programming APIs: A Study on GitHub and Stack Overflow

Carlos Zimmerle, Kiev Gama
Centro de Informática
Federal University of Pernambuco
Recife, Brazil
{cezl,kiev}@cin.ufpe.br

Fernando Castor*
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
castor@cin.ufpe.br

José Murilo Mota Filho
Centro de Informática
Federal University of Pernambuco
Recife, Brazil
jmsmf@cin.ufpe.br

ABSTRACT

Conventionally, callbacks and inversion of control have been the main tools to structure event-driven applications. Sadly, those patterns constitute a well-known source of design problems. The Reactive Programming (RP) paradigm has arisen as an approach to mitigate these problems. Yet, little evidence has been provided regarding the advantages of RP, and concerns have also arisen about the API usability of RP libraries given their disparate number of operators. In this work, we conduct a study on GitHub (GH) and Stack Overflow (SO) and explore three Reactive Extensions (Rx) libraries (RxJava, RxJS, and RxSwift) with the most GH projects to understand how much the vast Rx operators are being used. Also, we examine Rx SO posts to complement the results from the GH exploration by understanding the problems faced by RP developers and how they relate with the operators' frequencies found in open source projects. Results reveal that, in spite of its API size, the great majority of the Rx operators are actually being used (95.2%), with only a few, mostly related to RxJava, not being utilized. Also, we unveil 23 topics from SO with more posts concerning the Stream Abstraction (36.4%). Posts related to Dependency Management, Introductory Questions, and iOS Development figure as relevant topics to the community. The findings herein present can not only stimulate advancements in the field by understanding the usage of RP API and the main problems faced by developers, but also help newcomers in identifying the most important operators and the areas that are the most likely to be relevant for a RP application.

CCS CONCEPTS

• General and reference → Empirical studies; **• Software and its engineering → Data flow languages**.

KEYWORDS

Reactive Programming, API Usability, Mining Software Repositories

*Also with, Centro de Informática, Federal University of Pernambuco.

ACM Reference Format:
Carlos Zimmerle, Kiev Gama, Fernando Castor, and José Murilo Mota Filho. 2022. Mining the Usage of Reactive Programming APIs: A Study on GitHub and Stack Overflow. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524842.3527966

1 INTRODUCTION

Traditionally, event-driven applications, also called interactive or reactive applications (e.g., graphical user interface apps), have relied on callbacks and inversion of control as means to structure their logic [6, 12]. As those applications evolve, the code ends up becoming an asynchronous spaghetti with deeply-nested dependent callbacks, a problem often called callback hell or pyramid of doom [18]. As a result, event-based programs are known to be difficult to design, implement, and maintain [29]. The Reactive Programming (RP) paradigm was conceived to facilitate the construction of interactive applications through the use of dedicated abstractions [6, 19, 29]. Given the advantages of RP, many libraries and extensions have been incorporated in varying languages [6], including a specification for the JVM platform called Reactive Streams[1] that many libraries currently comply to (e.g., Akka Streams[2], Reactor[3], and RxJava). In spite of their clear benefits, data flow solutions like RP have been supported by little research evidence [27], and the necessity of carrying new research approaches like forum and repositories mining to better understand how RP is being used was previously pointed out [27] but never explored.

Reactive Extensions[4], also known as ReactiveX or Rx, is a popular family of libraries available for several programming languages. It was created for dealing with synchronous and asynchronous data as reactive streams. Surprisingly, Mogk et al. [21] report the presence of >450 variations of the Rx core operators[5] which contrasts with other libraries like Fran and REscala with ≈20 and ≈40 operators, respectively. This matches a familiar matter revolving around the widespread usage of combinators in functional programming which may impact the learnability and comprehension to newcomers [31]. In the same vein, Salvaneschi et al. [27] raised the issue that data flow languages may provide overspecialized operators. Some researchers [21, 31] believe that the focus should shift to designing small collection of core concepts, abstractions and operators, and ways to compose them rather than developing specializations. In

[1]https://www.reactive-streams.org/
[2]https://akka.io/
[3]https://projectreactor.io/
[4]https://reactivex.io/
[5]A complete list of the core operators is available at https://reactivex.io/documentation/operators.html.

this manner, RP can become more accessible to both newcomers and non-experts while still enabling the construction of complex programs [21, 31].

In light of the need for more studies aligned with the possibility of overspecialized operators in data flow solutions like RP, the present work attempts to investigate the usage of RP APIs. More specifically, we leverage the Rx API given its popularity, size, and polyglot nature with libraries for many languages. We believe that it is important not only to uncover if the operators are being used but also which kinds of problems RP users are facing and if there is any relation with the usage frequency of operators regarding those problems. We conducted a study on GitHub (GH) and Stack Overflow (SO) hoping to answer the following research questions:

- **RQ1.** How much are the Rx operators being used in open source projects?
- **RQ2.** What problems are reactive programming developers facing?
- **RQ3.** How do the operators present in the most relevant Stack Overflow questions and the usage frequency of Rx operators in open source projects relate?

To answer those questions, we explore the three Rx libraries with the most GH repositories: RxJava, RxJS, and RxSwift. Together, those libraries represent languages whose usage vary extensively, including UI, mobile, and web development.

The remaining of this paper is arranged as follows. Section 2 provides a brief background on RP. Section 3 delineates the methodology employed during the GH and SO mining, exposing the steps and decisions taken. Section 4 presents the results obtained through the methodology execution. Section 5 discusses the possible implications, followed by threats to validity (Section 6) and steps considered to mitigate them. Finally, Section 7 addresses the final remarks.

2 REACTIVE PROGRAMMING BACKGROUND

RP is a paradigm with origins on Functional Reactive Programming, primarily used to modelling animations [21], that was introduced to counter the problems (e.g., inversion of control, manual propagation of changes, and side effects) of imperative logic in interactive applications which contribute to difficult, buggy programs [6, 29]. It is formulated around dedicated abstractions [6, 27], behaviors (signals) and events (event streams), and those are in turn designed after three concepts [19]: (*i*) time-changing values, (*ii*) dependency tracking, and (*iii*) automatic propagation of updates.

RP has received much more attention of the programming language community and practitioners than from the software engineering community [29]. Aspects such as the higher composability of RP [28], and RP's improved comprehension when compared to the Observer pattern [31] have been explored. Recent efforts have focused on enabling RP in a distributed setting [12, 19, 20].

Reactive Extensions. Initially developed by Microsoft and made more well-known thanks to Netflix's successful use case [31], Rx stands as one of the most popular reactive libraries with implementations available for a variety of programming languages. Currently, Rx supports the event stream abstraction, called Observable, which represents a stream of value updates [6] and allows the composition of programs in a data flow style [27]. The computations are then constructed as a pipeline or series of stages, each one mostly consisting of some well-known functional operators (e.g., map, filter, etc) and callback-based or stream-like operators (e.g., counts and windowing) [9]. The side effects are usually pushed downstream to the stream consumer component, the observer object, which also controls the start of the stream execution as soon as a consumer is attached or connected (lazy evaluation nature).

3 METHODOLOGY

In this section we explain how we mined GitHub repositories (Section 3.1) and Stack Overflow questions and answers (Section 3.2) to address the three research questions. In general, the methodology applied to GH can be summarized as:

(1) Search for Rx repositories applying the defined star filter and store the information.
(2) Retrieve the repositories based on the stored information.
(3) Search for Rx operators within the download repositories **(RQ1)**.

Conversely, the overall SO methodology follows the following stages:

(1) Download Stack Exchange Data Explorer's data using Rx libraries' tags.
(2) Remove duplicates and consolidate the result files.
(3) Preprocess posts.
(4) Run LDA followed by topics' inference **(RQ2)**.
(5) Determine topics' relevance (popularity and difficulty) and search for operators among the posts **(RQ3)**.

Both Sections 3.1 and 3.2 access lists of operators of the different Rx libraries under analysis, and those lists were created by scraping official repositories of the distributions. The operators of RxJava and RxJS were extracted from their repositories on GitHub[6,7], while the ones from RxSwift were taken from the ReactiveX website[8]. By the time of the last scraping (September 27, 2021), the libraries were in the versions: 3.1.1 (RxJava), 7.3.0 (RxJS), and 5.1.1 (RxSwift). The scripts used for scraping are publicly available[9], as well as those for GH[10] and SO[11]. The scripts have also been archived through the Software Heritage[12] service.

3.1 GitHub Mining

We used the GitHub API, which accounts for almost 40% of the solutions used in the mining field [11] and allows to acquire repository data and metadata as well as commit messages, pull request information, etc. Researchers should take into account that many GH repositories are merely used to other concerns beside software development like storing personal data and (possibly inactive) repositories [17]. Social features like stars were used as a selection

[6]https://github.com/ReactiveX/RxJava/wiki/Operator-Matrix
[7]https://github.com/ReactiveX/rxjs/blob/master/docs_app/content/guide/operators.md
[8]https://reactivex.io/documentation/operators.html
[9]https://github.com/carloszimm/rx-scraping-msr22
[10]https://github.com/carloszimm/gh-mining-msr22
[11]https://github.com/carloszimm/so-mining-msr22
[12]They can be accessed through the URL https://archive.softwareheritage.org/browse/origin/ followed by their original URL.

Table 1: Information on the repositories using the five most used Rx libraries and the mined ones along with their star information, sorted by their total.

Library	Repositories			Mined Repositories			
	Total	Stars		Total	Stars		
		= 0	≥ 10		Min	Max	Median
RxJava	16,394	10,885	1,450	1,430	10	10,404	43
RxJS	16,380	12,740	818	797	10	16,835	30
RxSwift	5,505	3,833	402	401	10	13,644	37
RxKotlin	626	369	78	-	-	-	-
RxDart	493	274	73	-	-	-	-

Last update: January 7, 2022.

filter previously [34, 35] as an indicative of a repository's popularity and a favoring factor among developers. Thus, we used it to exclude potentially unimportant repositories. Following other approaches [16, 34], we set the exclusion threshold to consider repositories that have 10 or more stars. Table 1, column Repositories, outlines the total of repositories using the five most used Rx libraries along with the total of projects with zero stars and those with 10 or more stars. One can verify that repositories with 0 stars, and presumably low popularity, account for a big share of each library's total, supporting our choice of using stars as a filter.

The selection of the projects to be mined took into consideration the most used libraries. As depicted in Table 1, column Repositories, RxJava has the highest share of projects followed by RxJS and RxSwift. Considering the defined star filter (≥ 10) and aiming to obtain a significant sample size (>100), we decided to select the first three libraries (RxJava, RxJS and RxSwift). RxKotlin and RxDart were initially being considered to be included in the study but they would produce a small sample (<100 projects). Furthermore, RxKotlin takes most of its operators from RxJava, with only a small portion reserved for extension functions as delineated in its GitHub page[13]. Consequently, those libraries would not truly contribute to the study's objectives.

The actual sample size used corresponded to the de facto population of repositories with ≥ 10 stars for each investigated library. We selected the entire population willing not to incur in sampling errors and not to be unfair when defining a specific sample size for the three libraries when the population of dependent repositories has different dimensions.

The first step in our workflow was to look for Rx repositories, by using the name of the Rx libraries, along with the defined star filter. This search was then conducted by leveraging the 'search repositories' feature from the GitHub API. Afterwards, with that information saved in JSON files, we executed a script to download the repositories as tarball files. Given that it would not be feasible to store all the tarball files with our scripts for future replications, we stored the information about the downloaded files in a JSON file containing sufficient information to acquire them (e.g., a URL to download the file having the SHA1 hash of last commit already set) as well as details about the repository that file belongs to. This script took into consideration the exclusion of repositories that belong to official ReactiveX users[14] such as 'ReactiveX' or 'Reactive-Extensions'. The information about the repositories retrieved and processed is displayed in Table 1[15], column Mined Repositories. Having the downloaded repositories, a final script was executed to search for the Rx operators among the project files. The search was conducted by using a regular expression (regex), looking for operator invocations, either method (for method chaining pattern found in RxJava and RxSwift, for instance) or function (for function used with RxJS pipe method) calls. Before the actual counting for operators, a series of filters were used. First, a file extension filter to consider only files linked to each Rx libraries' language. Second, we checked whether the file had any mention to the Rx distribution considered at that moment, which would correspond to some kind of import to the library in the file. This filter was designed to minimize problems with a few functional operators often found in other constructs/libraries; however, the probability of false positives should be very low. Rx can wrap any type of value, even offers many generic creational methods for that purpose, and compose the logic as streams of those values. In this way, it reduces the need to invoking methods from other libraries (e.g., Java java.util.stream). In fact, we checked Java files, the language with more mined projects (Table 1), importing RxJava along with other Collection-like libraries (Java Streams[16], Eclipse Collections[17], Apache's CollectionUtils[18], and Guava's Collections2[19]). Results showed only 156 files out of 14,377, i.e., 1.09%. A random sample of 16 of those files (≈10%) yielded 62% of actual Rx operators, that is the great majority. Finally, we removed strings and comments of every file to avoid false positives from those constructs.

3.2 Stack Overflow Mining

The mining activities on SO were divided into two stages: topic modeling (Sections 3.2.1) and the definition of topics' relevance (3.2.2). Both stages use data collected via the Stack Exchange Data Explorer[20](SEDE), one of the many tools to acquire SO data already used in existing work [24, 32]. To have as much data as possible, we leveraged both questions and accepted answers, following Ahmed and Bagherzadeh [3] and Bajaj et al. [7]. We pulled questions, and their respective accepted answers, with the following tags: 'rx-java', 'rx-java2', 'rx-java3', 'rxjs', 'rxjs5', 'rxjs6', 'rxjs7', and 'rx-swift'; those are the tags directly related to the Rx libraries considered in this work: RxJava, RxJS, and RxSwift. Each query was executed separately for every tag and, afterwards, the results were combined, with the duplicated entries removed. A total of 47,404 records were fetched, with entries ranging from February 2011 to December 2021.

[13]https://github.com/ReactiveX/RxKotlin

[14]The list of official Rx GitHub users can be obtained by inspecting the URLs in https://reactivex.io/languages.html.

[15]We were not able to retrieve the following project due to file corruption: https://github.com/zwacky/game-music-player

[16]https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

[17]https://github.com/eclipse/eclipse-collections

[18]https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/CollectionUtils.html

[19]https://guava.dev/releases/23.0/api/docs/com/google/common/collect/Collections2.html

[20]https://data.stackexchange.com/

3.2.1 Topic Modeling. Topic modeling is a Natural Language Processing method based on the words' statistics that can be used to summarize documents through a set of topics [7]. Latent Dirichlet Allocation (LDA), on the other hand, is one of the most commonly-used topic-modeling techniques [10, 33]. It has been employed in many previous studies targeting SO data [1, 3, 24]. It allows a more manageable overview of a large corpora of documents as it may be impractical to do a manual inspection [24].

Posts Extraction. Data acquired through the SEDE comes in CSV format. It contains information about posts, such as titles, bodies, and types. For the purpose of this work, we initially decided to work with the posts' bodies. Working with the bodies gives us the opportunity to extract code snippets and help with other phases of the research (Section 3.2.2). Nonetheless, we observed that some posts, either questions or accepted answers, after going through data preprocessing, were becoming too small to offer any intuition about the context of the post for future analysis; examples include the question #65813454[21] and the accepted answer #41908578[22]. Therefore, based on Han et al. [13] and Han et al. [14], we mixed both title and body of the posts. For accepted answers, the titles of their respective questions were used to compose the effective post's text.

Data Preprocessing. A number of filters are commonly applied to documents before feeding them into the LDA for processing. We removed the following elements, based on [3, 13, 24, 33]: (1) code snippets, i.e. content inside <code>, <pre>, and <blockquotes>, (2) HTML tags, (3) Line breaks and sequence of whitespaces, (4) URLs, (5) one-letter words, (6) stop words, like *an* and *I*, (7) numbers, (8) punctuation marks, (9) non-alphabetical characters. Furthermore, we applied stemming (Snowball stemmer)[5, 10] to the remaining words (i.e., mapping the words to their root form) and removed common and uncommon words (i.e., words occurring in less than five posts or in more than 90% of those, respectively) [10]. After a few executions, we also decided to drop common SO words such as 'answer', 'question', 'help', and 'solut'.

LDA Execution. For the LDA execution, we leveraged the NLP[23] library, which offers many machine learning and natural language processing algorithms. One of the key points in LDA usage is to find the appropriate number of topics [1]. As noted by Han et al. [13], the exact quantity can impact the granularity of the result, yielding a too specific outcome, in case of a high number, or a too generic one, otherwise [1]. A common approach when deciding the number of topics is to vary this number [1, 24] and determine the most coherent result either by manual inspection [3, 24] or by some metric [1] like coherence. Following Rebouças et al. [24], we varied the number of topics between 10 and 35 and manually inspected the results. To help in the process, we also relied on the perplexity metric [33], which can be used to get some intuition about a possible good model fit. This metric tends become smaller as the number of topics grows [2], i.e., smaller values usually indicate better models. However, the correspondence of good model fit and human assessment do not always correlate [33]. Thus, we looked for outcomes yielding small improvements in perplexity with a different number of topics, but ultimately used the manual inspection to assess the results. Finally, we experimented with two combinations of values for hyperparameters: $\alpha = 50/k$, $\beta = 0.01$ [3, 26]—k denotes the number of topics—and $\alpha = \beta = 0.01$ [10, 25]. The first combination is a common one used in other studies but, conventional standards for parameters are not fit for GH and SO texts [33]. After a series of try-outs, the computed result with 23 topics, 1,000 iterations and $\alpha = \beta = 0.01$ showed the most coherent outcome.

Topic Inference. Topics produced by LDA correspond to a set of words and their proportion, with the name or label left to be inferred by who is applying the algorithm. To aid in this task, we resorted to the open card sorting technique [1, 3, 26], which consists of analyzing the top words of a given topic and inspecting posts chosen randomly that have the topic as their dominant one [3]. The first two authors were in charge of applying the technique by examining top 20 words and 15 random posts. Each examiner labeled the topics individually and jointly discussed results supported by a mediator (third author) to reach agreement when needed.

3.2.2 Defining Topic Relevance. In this study, we consider a topic as relevant based on its popularity and difficulty. To measure it, different metrics can be used. Popularity, for instance, can be calculated by taking the average value of three SO measures [1, 3]: (1) View, (2) Favorites, and (3) Score. Thus, a topic with high average view, favorites, and score is considered popular. Difficulty, conversely, has two metrics commonly employed [1, 3, 26]: (1) the percentage of questions with no accepted answer and (2) the median time it takes for an answer to be considered accepted; the time is calculated based on the creation dates found in the accepted answer and its respective question post [26]. As noted by Ahmed and Bagherzadeh [3], topics showing a high rate of questions without answers and taking more time to get accepted answers are intuitively harder. Hence, the aforementioned metrics are also exploited in the study.

Additionally, aiming to find operators' occurrences among the SO posts, we conducted a search throughout the questions and accepted answers used in Section 3.2.1. This search is carried out to delineate a correlation between the most relevant topics and the usage frequency of Rx operators (considered in Section3.1), the objective of RQ3. To access the code snippets, we extracted the contents inside the tags <code> of SO posts. As opposed to Section 3.2.1, we did not regard the <blockquote> and <pre> tags since <blockquote> is often used to include stack traces instead [25] and <pre> is usually applied to add formatting to <code>. Also, like Section 3.1, we removed comments and strings. The search in turn relied on a regex that either looked for the operator name or the operator invocation. The examination of only the operator name is necessary since, many times, text inside the <code> tag is used only to highlight a construct without actually showing its usage (e.g., post #40811273[24]).

4 RESULTS

This section is organized according to research questions (RQ1-RQ3).

[21]https://stackoverflow.com/q/65813454
[22]https://stackoverflow.com/a/41908578
[23]https://github.com/james-bowman/nlp
[24]https://stackoverflow.com/questions/40811273

4.1 RQ1: How much are the Rx operators being used in open source projects?

To elaborate the results, we relied on the usage frequency of the operators from RxJava, RxJS, and RxSwift as detailed in Section 3.1. Figure 1 presents the percentage of operators' utilization according to their library. We can observe that the majority of operators are actually in current use. The greatest exception among the three libraries was RxJava presenting 94.1% (223 operators) of usage but 5.9% (14) of its 237 operators not actually being used in the GitHub projects of our sample. RxJS and RxSwift, on the other hand, presented 100% and 98.5% of employment of their 113 and 66 operators, respectively. Nonetheless, in general, the libraries showed a good measure of utilization. This fact is even more clear when combining the operators (merging those alike) and their frequencies from all three libraries and computing the percentage of utilization: 95.2% of general utilization. Thus, although Rx provides a great number of operators scattered through its different libraries, those operations showed a very considerable rate of utilization. From the 14 non-used operators of RxJava, we could notice that half were operators related to Java's concurrency API *CompletionStage*[25] whose support was added in RxJava 3 (released on February, 2020). This may indicate that either the developers did not have time to use the feature yet or did not find a useful situation to apply it.

Figure 2, 3, and 4 depict the most and least frequently used operators in RxJava, RxJS, and RxSwift, respectively. For simplicity, we included only the 15 most and least utilized ones and removed those not in use. The complete list according to their frequency is available online. By inspecting the charts with the most used operators, one can perceive the occurrence of *subscribe* as one of the most utilized operators in the three libraries. This correlates to its importance in controlling the stream lifecycle. The stream becomes indeed active only when subscribed, given its lazy evaluation principle. Curiously, *map* is the most intensively used operator in RxSwift. Still, *map* is certainly an important transforming operator, and that can be evidenced by its presence in the list of most frequently used operators of the three libraries. Along with *map*, one can perceive the presence of other common functional operators like *any*, *filter*, and *reduce*. Concerning the creation operators, we can note that *just* or its equivalent *of* is one of the most present. It allows the creation of a one-element stream and can be useful for the construction of business logic that expects the emission of a single element in their composition. Apart from RxJava, there were few operators related to error handling/testing. In summary, the most frequently used operator comprises all categories[26] of operators such as creation (*of*, *from*), transforming (*map*, *flatMap*), filtering (*take*, *takeUntil*), combining (*merge*, *zip*), among others. A pattern that we could noticed is that the majority of the most used operations are largely composed of Rx core operators (e.g., *concat*, *filter*, *map*, *etc.*). The least frequently used, on the contrary, are mostly formed by library-specific variants (e.g., *concatMapSingle*, *concatMapMaybe*, *mapWithIndex*, etc.) which comprises all types of operators. An exception in the set of least utilized ones is the presence of core operators *buffer* and, specially, *window* (and

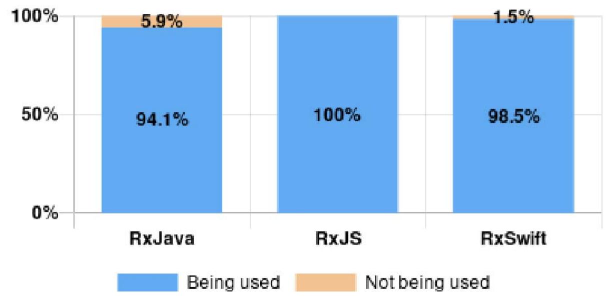


Figure 1: Percentage of operators' utilization per Rx library.

Table 2: Average usage of the 15 most frequently used operators per analyzed repository.

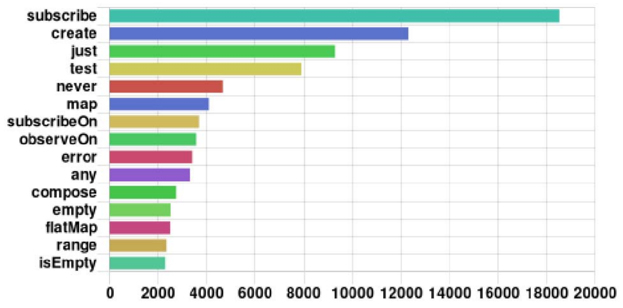
RxJava	RxJS	RxSwift
subscribe (13.0)	subscribe (16.7)	map (62.0)
create (8.6)	map (14.7)	subscribe (36.6)
just (6.5)	of (5.3)	create (19.9)
test (5.5)	filter (5.0)	just (19.2)
never (3.3)	concat (3.6)	combineLatest (10.4)
map (2.8)	from (2.7)	empty (10.2)
subscribeOn (2.6)	take (2.2)	zip (8.8)
observeOn (2.5)	tap (2.2)	flatMap (8.5)
error (2.4)	find (2.1)	filter (6.9)
any (2.3)	switchMap (1.6)	observeOn (6.7)
compose (1.9)	merge (1.5)	merge (6.0)
empty (1.8)	reduce (1.4)	distinctUntilChanged (4.3)
flatMap (1.7)	mergeMap (1.2)	startWith (4.0)
range (1.6)	takeUntil (0.9)	takeUntil (3.2)
isEmpty (1.6)	fromEvent (0.9)	from (3.1)

Note. Average usage shown in parentheses.

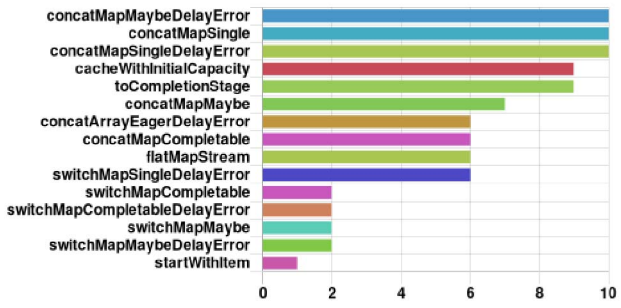
some of its variants such as *windowTime*, *windowToggle*, and *windowWhen*). These correspond to an important class of operators since some computations require the accumulation of stream elements before execution. Table 2 presents a different perspective for the most used operators in the three libraries. It shows the average usage per analyzed repository. Those statistics indicate that even thought the operators have shown relative high frequencies, they are probably being used more in some projects than others (i.e., in an irregular way). A final worth observation is the scale of the least used RxJava operators. Its 15 least used operators did not exceed the limit of 10 usages, which is likely linked to its extensive API. However, when collecting operators with ≤50 uses throughout the analyzed projects, we could notice that RxJS presented the greatest percentage, corresponding to ≈32% of its operators against ≈22% from RxJava and only ≈7.5% from RxSwift.

The great majority of Rx operators are being utilized in the libraries RxJava (94.1%), RxJS (100%), and RxSwift (98.5%). This percentage comes to 95.2% when merging the operators and their usage frequency from all three libraries. ***Finding 1***

[25]https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html
[26]Categories based on https://reactivex.io/documentation/operators.html.

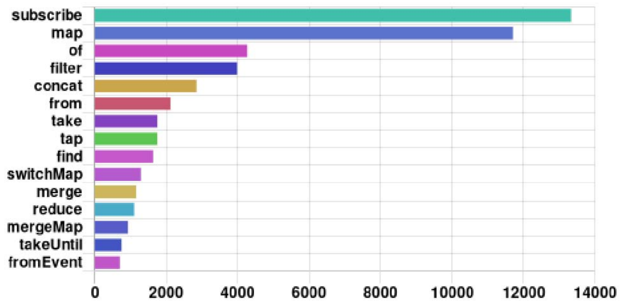


(a) 15 most used

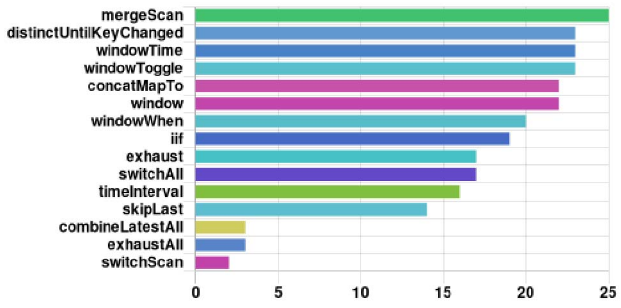


(b) 15 least used

Figure 2: The most and least frequently used operators of RxJava.

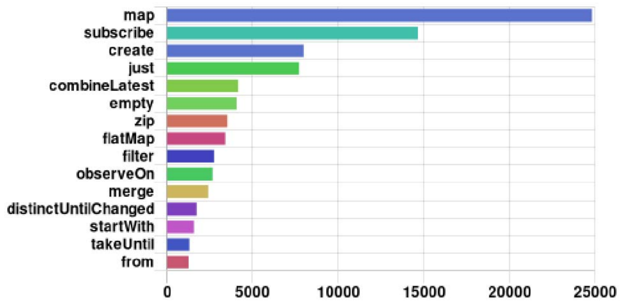


(a) 15 most used

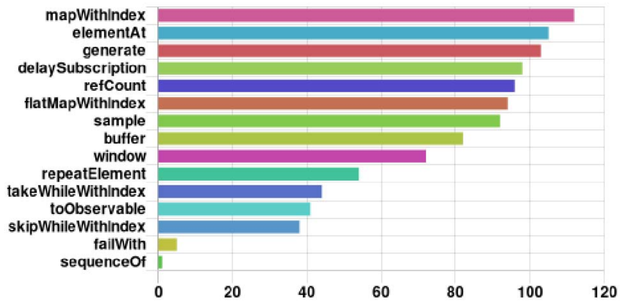


(b) 15 least used

Figure 3: The most and least frequently used operators of RxJS.



(a) 15 most used



(b) 15 least used

Figure 4: The most and least frequently used operators of RxSwift.

Only RxJava presented more than one operator not being utilized (≈6%). ***Finding 2***

subscribe appears as the most frequently used operation in two Rx libraries: RxJava and RxJS. Yet, it is the second most used in RxSwift, only behind the map operator. ***Finding 3***

Functional operators, like any, filter, and map, figure among the most used operators in all libraries. ***Finding 4***

The most used operators mainly comprise core operators, while the least used ones are essentially composed of core variants. ***Finding 5***

All 15 least used operators of RxJava showed less than or equal to 10 usages. However, RxJS presented the greatest percentage of operators with ≤50 usages (≈32%). ***Finding 6***

4.2 RQ2: What problems are reactive programming developers facing?

As denoted in Section 3.2, we used the LDA topic modeling technique to uncover the problems, in a topic format, that reactive programming developers are facing. Table 3 presents the 23 generated topics with the inferred labels and sorted by their number of posts. From the result, we can observe that the generated topics truly correspond to important matters that reactive programmers face daily. For example, the first two topics with the most posts are related to stream abstraction. We can also note presence of *Concurrency*, an important intrinsic concept in reactive programming given that its pipeline model allows the exploration of an asynchronous and non-blocking execution and the efficient use of threads [9], either directly (as by schedulers in RxJava) or indirectly. One can also visualize elements revolving UI, a field from where most of the reactive programming research came from by the developing of the Fran language [6]. We grouped those topics under nine categories and briefly discuss the three with the largest share of posts.

Stream Abstraction (6 topics, 36.4% posts). This category encompasses six topics related to the stream abstraction (i.e., the main resource to structure the reactive logic): *Stream Manipulation*, *Stream Creation and Composition*, *Stream Lifecycle*, *Timing*, *Multicasting*, and *Control Flow*. Together, their posts account for 36.4% of the total of analyzed posts with the first two topics showing the biggest share as shown in Table 3. Under the topic *Stream Manipulation*, for instance, there were many question related to `Observable` (main stream type in Rx) handling like converting it, accessing and passing values to it, returning it, among others. For example, ℚ48601357 asks *"...convert Observable<string[]> into Observable<string>"* while ℚ56610867 deals with *"A more succinct way to conditionally chain multiple observables."* Having this topic as the one with the higher number of posts is probably due to the shift that developers should face while structuring their code as compositions of streams. The second topic with the most posts, *Stream Creation and Composition*, is very related to the first one and surprisingly showed many operators in their LDA top words, such as 'combin', 'merg', and 'combinelatest'. ℚ38067532 is a typical example of questions found in this topic where the author asks about *"Combining two different observables."* The remaining topics cover other matter intrinsic to streams, like the use of time handling/control flow operations, subscribing and unsubscribing (i.e., lifecycle), the different types of stream "temperature" (hot or cold[27]), or the proper usage of Subject, an alternative to Observable that is hot in nature (multicasting).

Application Development (5 topics, 21.2% posts). This category combines problems related to application development in general, totaling five topics. The addition of those topics' posts yields a share of 21.2% of all study's posts. From the posts in this category, we have 42.2% about Web development (considering *Web Development* and *State Management and JavaScript* together) and 40.8% about mobile development (*Android Development* and *iOS Development*). Most of the web development topics revolves around the Angular framework which is expected given that Angular both use RxJS internally and also makes it available to be utilized as part of its library[28]. Others, conversely, deal with state management, an important matter in nowadays JavaScript frameworks. In this regard, we have many mentions to the Redux-Observable (a Redux middleware) and, specially, NgRx (state management for Angular based on RxJS). In the mobile development, more posts linked to Android than iOS appeared, even though we did not include any SO tags related to RxKotlin or RxAndroid (RxJava bidings for Android). In the iOS posts, we could observe a prevalence of questions related to patterns embracing view models like MVVM (Model-View-ViewModel). The Android ones, on the contrary, included many questions about using RxJava with some framework or library like Retrofit (an HTTP client). The remaining posts within this category, curiously, related to dependency management problems like in ℚ45516375 where the author complains about *"Problems*

Table 3: Topics ordered by their number of posts.

Topic	# Posts	% Posts	Category
Stream Manipulation	6384	13.5	Stream Abstraction
Stream Creation and Composition	4872	10.3	Stream Abstraction
Array Manipulation	3146	6.6	Programming
Web Development	3049	6.4	Application Development
Data Access	2680	5.7	Persistence
Android Development	2551	5.4	Application Development
Concurrency	2251	4.7	Concurrency
HTTP Handling	2243	4.7	Networking
Stream Lifecycle	2141	4.5	Stream Abstraction
Error Handling	1825	3.8	Reliability
Timing	1813	3.8	Stream Abstraction
UI for Web-based Systems	1791	3.8	User Interface
Dependency Management	1710	3.6	Application Development
Typing and Correctness	1578	3.3	Programming
iOS Development	1536	3.2	Application Development
Multicasting	1305	2.8	Stream Abstraction
REST API Calls	1231	2.6	Networking
Testing and Debugging	1219	2.6	Reliability
State Management and JavaScript	1183	2.5	Application Development
Input Validation	1087	2.3	User Interface
Control Flow	735	1.6	Stream Abstraction
General Programming	588	1.2	Programming
Introductory Questions	486	1.0	Basics

Total of Posts = 47,404.

[27]Cold streams are those that produce data based on each subscription. Hot ones, on the other hand, do not depend on subscription to emit values and their values are normally produced from an outside source.

[28]https://blog.angular-university.io/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/

with .maps and import 'rxjs/add/operator/map'" or ℚ46692177 with the issue *"RXJS Observable missing definitions."*

Programming (3 topics, 11.2% posts). Three topics are classified under this category: *Array Manipulation*, *Typing and Correctness*, and *General Programming*. The three topics represent 11.2% of the investigated posts, with *Array Manipulation* comprising most of the posts (59%), as expected. Dealing with stream manipulation often requires the conversion to or from an array (e.g., the *toArray* operator from RxJS) or even the accumulation of elements in arrays (the case of *buffer* operator) given that streams can be seen as a sequence of events. In ℚ58657165, for example, the user complains *"Can not get array of Observables."* Other questions in the *Array Manipulation* revolved around manipulating collections of Observables (streams) or the use of common collection-like functions (e.g., 'filter'). The posts under *Typing and Correctness* entered this category since they mostly show problems of type systems like *"Property 'forEach' does not exist on type 'Object'"*(ℚ46352289) or *"Type 'Observable< | T>' is not assignable to type 'Observable<T>'"*(ℚ45273084). The rest of the posts under this category were classified as *General Programming*, given that they discuss primarily about handling of common programming constructs like objects, variables, null values, etc.

Reactive developers ask about a multitude of topics, like Stream Manipulation, Web Development, Concurrency Networking, etc., but with higher interest in problems regarding the stream abstraction. ***Finding 7***

The topics with more posts concern the stream abstraction, having Stream Manipulation (13.5%) and Stream Creation and Composition (10.3%) occupying the first and second places. ***Finding 8***

Developers have asked less about basic matters as can be verified by the topics with fewer posts: General Programming (1.2%) and Introductory Questions (1%). ***Finding 9***

4.3 RQ3: How do the operators present in the most relevant Stack Overflow questions and the usage frequency of Rx operators in open source projects relate?

We considered a topic as relevant based on its popularity and difficulty. Table 4 shows the popularity of the topics according to the average views, favorites, and scores of their questions. Table 5 exhibits the difficulty of the topics according to their percentage of questions without an accepted answer and median time taken to receive an accepted answer. Tables 4 and 5 are respectively sorted by the average views and the percentage of questions without an accepted answer.

The observation of Table 4 reveals that *Dependency Management* has the highest average number of views, while *Introductory Questions* shows the greatest average favorite and score. Hence, those topics are among the most popular ones. In contrast, *Data Access* exhibits low average view, favorite, and popularity, thus it is among the least popular questions. The topics in Table 5, in turn, show *Dependency Management* (highest rate of questions without accepted answer) and *iOS Development* (greatest median time) among the most difficult topics to answer. Conversely, *Array Manipulation* and

Table 4: Topics' Popularity.

Topic		Average	
	Views	Favorites	Scores
Dependency Management	**3453.6**	0.7	3.8
Web Development	2707.6	0.6	2.2
Stream Manipulation	2641.4	0.8	3.5
Typing and Correctness	2527.6	0.3	2.3
Stream Lifecycle	2503.1	1.1	4.2
Introductory Questions	2442.5	**2.2**	**6.5**
Multicasting	2369.9	1.1	3.9
Error Handling	2250.2	0.6	2.6
Control Flow	1923.0	0.4	2.2
Array Manipulation	1803.7	0.3	1.4
Android Development	1793.0	0.8	2.7
General Programming	1722.4	0.4	2.2
UI for Web-based Systems	1659.8	0.4	1.7
iOS Development	1634.3	0.5	1.9
Input Validation	1489.7	0.3	1.4
HTTP Handling	1470.8	0.5	1.7
Timing	1454.7	0.4	2.1
Stream Creation and Composition	1421.9	0.5	2.3
REST API Calls	1341.5	0.4	1.3
Concurrency	1308.9	0.7	2.6
Testing and Debugging	1237.9	0.3	1.8
State Management and JavaScript	1137.8	0.4	1.6
Data Access	1102.5	0.3	1.3

Web Development are the easiest topics, with the least percentage of questions with no accepted answer and median time in hours, respectively.

Given that *Dependency Management* is classified as both popular and difficult, there are three most relevant topics: *Dependency Management*, *Introductory Questions*, and *iOS Development.* Figure 5 shows the similarities between the operators gathered through the posts of those topics and the ones collected in GH projects (Section 3.1) according to the order of their frequencies. This comparison was calculated based on the complement of Hamming Distance [15]. We can observe that very few operators share similarity when considered their position based on their frequencies. By looking at the sample of the 15 most used operators in both the most relevant topics and the GH projects and disregarding their order of appearance, we can notice many more matches as shown in Figure 6. Those matches can be observed in Table 6. We can also notice many of the most frequent ones out of the 15 most used in Figure 2, 3, and 4 also appearing in Table 6. Unsurprisingly, *subscribe* shows high frequencies as in the case of Introductory Questions, with 200 occurrences for RxJava posts. Well-known functional operators are also present, with some exhibiting great frequencies like *map* with 811 (RxJS) and 833 (RxSwift) for Dependency Management and iOS Development, respectively. Actually, this high frequency observed for *map* in RxSwift corroborates the findings in Section 4.1, giving more credibility, where *map* was also the most utilized operator in RxSwift. Also, following Section 4.1 findings, most of the operators are the ones considered as core; exceptions include, for example, the creation operator *fromEvent* and the utility *test* method.

Table 5: Topics' Difficulty

Topic	w/o Acc. Answer(%)	Median Time(hr)
Dependency Management	**50.3**	1.2
Testing and Debugging	49.0	3.3
Multicasting	48.1	2.7
iOS Development	47.8	**5.5**
Android Development	47.8	2.3
Concurrency	47.1	3.5
Data Access	46.5	1.9
UI for Web-based Systems	46.5	0.9
HTTP Handling	46.2	1.2
Error Handling	46.0	1.6
REST API Calls	44.2	1.0
Input Validation	43.9	1.2
Introductory Questions	43.3	4.8
Stream Lifecycle	43.3	1.5
Web Development	43.2	0.5
Control Flow	42.2	1.4
State Management and JavaScript	41.9	2.2
Timing	41.0	2.2
Stream Creation and Composition	38.4	1.8
Stream Manipulation	37.7	0.8
Typing and Correctness	37.3	0.8
General Programming	37.0	1.2
Array Manipulation	35.7	0.8

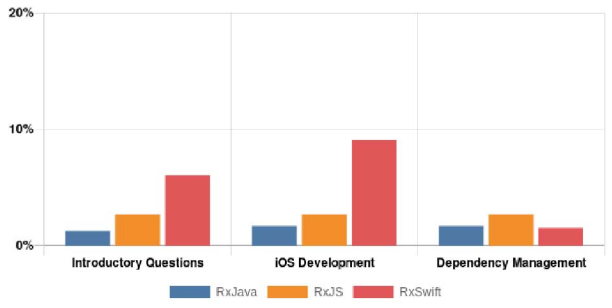


Figure 5: Similarity between the operators from the most relevant SO topics and the ones from open source projects based on their frequency position.

Dependency Management and Introductory Questions are among the most popular topics with the former having the greatest number of views and the latter presenting the highest favorites and score, on average. Problems regarding Data Access figure among the least popular. ***Finding 10***

Posts concerning Dependency Management are amongst the most difficult questions. iOS Development also figure as one of the most challenging topics, whereas Array Manipulation and Web Development appear among the easiest ones. ***Finding 11***

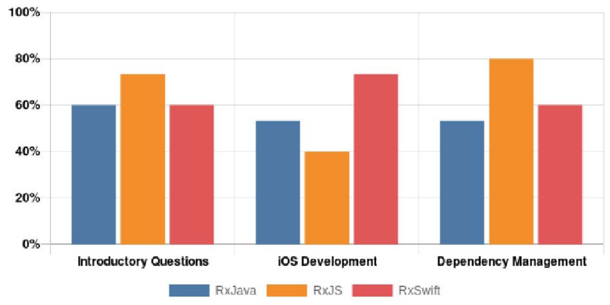


Figure 6: Similarity between the 15 most frequently used operators in the most relevant SO topics and the 15 ones from open source projects regardless of their frequency position.

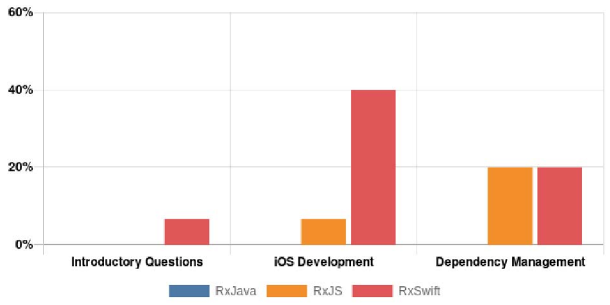


Figure 7: Similarity between the 15 least frequently used operators in the most relevant SO topics and the 15 ones from open source projects regardless of their frequency position.

Although the usage frequency of operators of the most relevant SO topics and the GH projects do not share much similarity when comparing the order of appearance according to their frequency, the most frequently used operators of both sources showed a high percentage of similarity when considering only their matches regardless of their frequency position. ***Finding 12***

The majority of the most frequently used operators in SO posts which share similarities with the most frequently used operators in GH repositories is also mostly composed of core operators. ***Finding 13***

The least frequently used operators in SO showed small percentage of similarity when compared to the least utilized ones in GH projects. ***Finding 14***

5 IMPLICATIONS

Developers. The findings delineated in Section 4.1 and 4.3 can serve as a start point for those novice developers that are trying RP. As noted, the great majority of the most used operators both in open source projects and SO forum are mostly comprised of common core operators. So, concentrating their efforts in those operators could facilitate their learning path, with support from both platforms.

Table 6: Operators most frequently used in the most relevant SO topics matching the most frequently used operators in GH projects.

Rx		Operators	
Library	Dependency Management	Introductory Questions	iOS Development
RxJava	test (66)	subscribe (200)	subscribe (110)
	subscribe (65)	map (95)	error (35)
	create (37)	create (72)	create (32)
	error (27)	flatMap (63)	observeOn (27)
	map (23)	just (39)	subscribeOn (26)
	flatMap (18)	error(35)	map (17)
	any (16)	subscribeOn (27)	flatMap (13)
	observeOn (13)	test (26)	test (10)
		observeOn (21)	
RxJS	from (2311)	subscribe (159)	subscribe (71)
	map (811)	map (154)	map (56)
	subscribe (649)	from (124)	from (49)
	of (644)	of (82)	of (36)
	merge (263)	switchMap (45)	filter (13)
	mergeMap (189)	filter (44)	tap (9)
	find (185)	merge (33)	
	filter (159)	fromEvent (31)	
	switchMap (116)	take (29)	
	take (107)	mergeMap (23)	
	tap (91)	concat (15)	
	fromEvent (86)		
RxSwift	from (15)	map (24)	map (833)
	flatMap (10)	subscribe (21)	subscribe (733)
	map (10)	create (19)	create (295)
	subscribe(6)	filter (16)	filter (291)
	combineLatest (3)	flatMap (12)	flatMap (290)
	observeOn (2)	just (4)	just (167)
	create (1)	empty (3)	from (139)
	empty (1)	from (3)	combineLatest (114)
	filter (1)	startWith (3)	observeOn (95)
			empty (77)
			distinctUntilChanged (71)

Note. Operator's frequency in the SO posts shown in parentheses.

Maintainers. API call frequencies offer the opportunity for API designers to comprehend the effects of API deprecation and direct efforts [36]. Although, the Rx operators demonstrated a great usage (Section 4.1), some actually showed low frequency. Among the RxJava operators, for instance, there were 14 (5.9%) with no usage and, of the 15 with lowest usage, all demonstrated ≤10 calls. Besides, 32% and 22% correspond to the percentage of RxJS and RxJava operations with ≤50 calls, respectively. This contrasts to others operators displaying much more than 1,000 calls amongst the most used ones (Figure 2, 3, and 4). This give rise to a possible consideration of API reduction and the suggestion provided by some researchers of shifting focus from specialization to core concepts [21, 31].

As exposed in Section 4.3, *Dependency Management* topic was both classified as popular and difficult. This prompts a possible future evaluation to investigate why a topic that represents only 3.6% of the posts is receiving both classifications. Among the posts we could perceive, as noted in Section 4.2, many problems related to the building process, dependency handling, imports, etc. Having *Introductory Questions* as one of the most popular topics is possibly a sign that newcomers are showing interested in RP but may be encountering problems in the process of understanding it. In ℚ36535716, for instance, the user states "I read a ton of literature about the Rx and, on the one hand, everything is clear, but on the other hand nothing is clear." In fact, Salvaneschi et al. [31] detailed three points against RP through a qualitative analysis that may be very linked to this SO topic: learning curve, higher level of abstraction (reliance on the runtime), and connection to functional programming. Giving that RP slightly changes the way that most programmers are used to design their programs, it is paramount to provide good documentation and resources.

Researchers. The findings discovered in Section 4.1 show that even though Rx owns an API that is largely extended by its libraries, i.e., by the addition of many variants, we can conclude that developers are actually using a large portion (>90%) of those operators, and from those, the majority of the most used ones is composed by core operators (e.g., *just*, *map*, and *filter*). Hence, by visiting the question suggested by Salvaneschi [27] "Do data flow languages provide a 'simple enough' solution for the common case without excessive proliferation of overspecialised operators?", on the one hand, we can assume that the operators provided for reactive programmers through the Rx library are offering a "simple enough" solution at a certain extent given that only a few operations have shown no usage and the most used ones are concentrated at the core operators both in open source projects and SO forum. On the other hand, regarding the overspecialization of operators, the core operators actually present a considerable API surface (≈70-80 operators [21, 27]), and APIs like RxJava exhibited low frequencies (≤10) for its least used operations and no frequency at all for 14 operators (5.9%). Thus, further understanding from a user-centered point of view might shed more light on this concern and help fully answer the proposed question. In the same vein, the topic *Introductory Questions* figured as a popular one that could, among various reasons (e.g., documentation [23], learning curve and relation with functional programming [31], or diverse API design decisions), also be related to this API surface. Therefore, we strongly believe that more API usability studies are needed, with data collected from both novice and experienced developers, specially when only few studies have been executed in the area [27].

An important observation taken from Table 5 is the presence of the *Testing and Debugging* with the second highest percentage of question without an accepted answer. Moreover, from the mining process, there seems to be few operators in this area. This prompt us to question if this topic is showing relevance due to a lack of dedicated facilities for testing and debugging or difficulty for using the API, and, thus, preparing testable code. In theory, testing should not be such an issue given that there is a separation of concerns of the main entities in the stream model (i.e., producers and consumers of data and, in-between, a pipeline of pure functions) which could facilitate the process. Mogk et al. [21], for instance, report no specific obstacles with testing in REScala. Besides, Rx offers schedulers and marble diagrams for testing purposes to give more control over time and represent more easily stream events, respectively. However, for effective testable code in the stream model, it may require a certain discipline from developers (e.g., decouple stream parts properly, keep pipeline functions free of side effects, etc.). For debugging reactive programming, conversely, the unsuitability of usual tools has already been recognized [8, 22, 30]. Salvaneschi and Mezini [30] proposed a technique and tool (Reactive Debugging and Reactive Inspector, respectively) to help visualizing the dataflow of the application, with extensions to provide live programming [22]. Cycle.js[29], a JavaScript functional and reactive framework, also provides a similar tool for Internet browsers that can be used to view the dataflow graph and, as a result, help the debugging and metal model process. The lack of integration with usual debugging tools of common IDEs and browsers was actually a downside for adoption of some RxJS debugging tools as pointed by Alabor and Stolze [4]. Consequently, we believe that the area of *Testing and Debugging* constitutes a rich and prominent direction for future exploration and investigation.

6 THREATS TO VALIDITY

Internal Validity. Internal validity may be described as aspects that could influence our outcomes [1] or possible mistakes in execution and experiments [33]. To search for Rx operators, we looked through every file identifying by specific file extensions (according to the language of the analyzed Rx distribution) and relied on the use of regular expression (regex). By using a regex, this could indicate a threat concerning some Rx operators that may have the same name as well-known functional operators like 'filter' and 'map'. To reduce this threat, we checked if every inspected file had any mention to the investigated libraries (e.g., rxjava or rxjs) which would correspond to some import of library's package in that specific document. Besides, we examined RxJava files, the library with most mined projects, to verify how much false positives were being introduced. Given that Rx can wrap any type of values as streams, the number of false positives revealed to be very low, only accounting for a tiny percentage of files (1.09%) in which a 10% sample showed majoritively Rx operators (62%). However, future replications should strive to use alternative methods that diminish false positives like the semantic tool explored by Xu et al. [35]. Finally, we also removed both comments and strings, so code snippets inside those constructs would not be counted.

According to Abdellatif et al. [1], the choice of the optimal amount of topics for the LDA algorithm could constitute a threat since it is recognized as a difficult task and directly impacts the quality of the generated LDA topics. To mitigate that possible threat, we followed Abdellatif et al. [1] and Rebouças et al. [24] by experimenting with a range of topics; besides, we resorted to the Perplexity metric to aid the manual definition of the optimal number of topics. Still according to Abdellatif et al. [1], the inference of the resulting LDA topics could represent a threat given its subjectiveness. We countered this threat by having more than one author evaluating the topics using the open card sort technique and trying to reach some agreement afterwards, similar to the studies [1, 3].

[29]https://cycle.js.org/

External Validity. This validity refers to the generalization of our discoveries [1, 10, 33]. The present work focused on two prominent platforms, GitHub and Stack Overflow. Their widely usage in the development setting gives us a certain confidence about our findings, though other sources like alternative forums or hosting code platforms could improve the final result. Thus, we believe that further incorporation of other sources can complement our findings as well as the inclusion of surveys and qualitative studies. Nonetheless, we tried to include as much data as possible, working with all data in GH and SO regarding the chosen Rx libraries.
Construct Validity. Construct validity is about the fitness of the metrics used in the evaluation [10, 33]. Treude and Wagner [33] report that the Perplexity metric and human assessment do not oftentimes correlate. Thus, we merely used it as an aid to identify the number of topics, and we mostly relied on the manual inspection carried out by the first two authors, similar to Han et al. [13]. Future works may incorporate other better metrics (e.g., coherence).

The metrics used to determine the popularity or difficulty of the SO topics could represent a threat as pointed by Abdellatif et al. [1]. In this regard, we resorted to metrics used in previous studies [1, 3, 26] as a mean to counter the threat. Yet, the difficulty aspect, despite being explored in other studies, could actually indicate other circumstances such as lack of popularity or interest in the topic from the community; thus, the clarification of the different possible meanings for difficulty may be a relevant research endeavor.

7 CONCLUSION

In this paper, we evaluated to what extent the reactive operators are being used. To accomplish it, we leveraged the ReactiveX API by considering the three Rx libraries with the most GitHub projects: RxJava, RxJS, and RxSwift. To complement our findings, we also conducted a mining in the Stack Overflow to uncover which problems reactive programmers are most asking and how they relate to the usage frequencies of Rx operators. Our findings showed that, despite its size, the majority of the Rx library API is in use, with only a few operators either not being used or with low usage frequencies. Also, we inferred 23 topics that reactive programmers have discussed and grouped them into nine categories. Results have shown that the greatest focus of the SO posts have regarded the stream abstraction, with *Introductory Questions*, *iOS Development*, and *Dependency Management* among the most popular and difficult areas. Furthermore, the operators most frequently used in open source projects and SO forum share great similarities, giving credits to our findings. In turn, the presented findings and implications can not only help developers, but also maintainers and researchers. Additionally, the topic *Testing and Debugging* demonstrated to be an interesting area for further investigation. Potential avenues of future work include different RP API usability researches to complement the delineated implications of the present study and the execution or comparison of similar studies with other RP, dataflow-like libraries.

ACKNOWLEDGMENTS

This work is partially supported by INES (www.ines.org.br), CNPq grant 465614/2014-0, FACEPE grants APQ-0399-1.03/17 and APQ/0388-1.03/14, CAPES grant 88887.136410/2017-00.

REFERENCES

[1] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. 2020. Challenges in chatbot development: A study of stack overflow posts. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 174–185.

[2] Amritanshu Agrawal, Wei Fu, and Tim Menzies. 2018. What is wrong with topic modeling? And how to fix it using search-based software engineering. *Information and Software Technology* 98 (2018), 74–88.

[3] Syed Ahmed and Mehdi Bagherzadeh. 2018. What do concurrency developers ask about? a large-scale study using stack overflow. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[4] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. 15–24.

[5] Miltiadis Allamanis and Charles Sutton. 2013. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 53–56.

[6] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Computing Surveys (CSUR)* 45, 4 (2013), 1–34.

[7] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2014. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 112–121.

[8] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging data flows in reactive programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 752–763.

[9] Jonas Bonér and Viktor Klang. 2017. Reactive programming versus reactive systems. *Dosegljivo: https://www. lightbend. com/reactiveprogramming-versus-reactive-systems.[Dostopano: 23. 08. 2017]* (2017).

[10] Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. 2015. Latent Dirichlet allocation: extracting topics from software engineering data. In *The art and science of analyzing software data*. Elsevier, 139–159.

[11] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2016. Findings from GitHub: methods, datasets and limitations. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 137–141.

[12] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An update algorithm for distributed reactive programming. *ACM SIGPLAN Notices* 49, 10 (2014), 361–376.

[13] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. 2020. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering* 25, 4 (2020), 2694–2747.

[14] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. 2017. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 125–136.

[15] Hadi Hemmati and Lionel Briand. 2010. An industrial investigation of similarity measures for model-based test case selection. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 141–150.

[16] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. 2020. Learning from, understanding, and supporting DevOps artifacts for Docker. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 38–49.

[17] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[18] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. 1–9.

[19] Alessandro Margara and Guido Salvaneschi. 2014. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. 142–153.

[20] Alessandro Margara and Guido Salvaneschi. 2018. On the semantics of distributed reactive programming: the cost of consistency. *IEEE Transactions on Software Engineering* 44, 7 (2018), 689–711.

[21] Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Reactive programming experience with rescala. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. 105–112.

[22] Ragnar Mogk, Pascal Weisenburger, Julian Haas, David Richter, Guido Salvaneschi, and Mira Mezini. 2018. From debugging towards live tuning of reactive applications. In *2018 LIVE Programming Workshop. LIVE*, Vol. 18.

[23] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of API usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 5–14.

[24] Marcel Rebouças, Gustavo Pinto, Felipe Ebert, Weslley Torres, Alexander Serebrenik, and Fernando Castor. 2016. An empirical study on the usage of the swift programming language. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 634–638.

[25] Leonardo Jiménez Rodríguez, Xiaoran Wang, and Jilong Kuang. 2018. Insights on apache spark usage by mining stack overflow questions. In *2018 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 219–223.

[26] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering* 21, 3 (2016), 1192–1223.

[27] Guido Salvaneschi. 2016. What do we really know about data flow languages?. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. 30–31.

[28] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on Modularity*. 25–36.

[29] Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. 2015. Reactive programming: A walkthrough. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 953–954.

[30] Guido Salvaneschi and Mira Mezini. 2016. Debugging for reactive programming. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 796–807.

[31] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2017. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1125–1143.

[32] Amjed Tahir, Jens Dietrich, Steve Counsell, Sherlock Licorish, and Aiko Yamashita. 2020. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology* 125 (2020), 106333.

[33] Christoph Treude and Markus Wagner. 2019. Predicting good configurations for github and stack overflow topic models. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 84–95.

[34] Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. 2020. An empirical study of quick remedy commits. In *Proceedings of the 28th International Conference on Program Comprehension*. 60–71.

[35] Yisen Xu, Fan Wu, Xiangyang Jia, Lingbo Li, and Jifeng Xuan. 2020. Mining the use of higher-order functions. *Empirical Software Engineering* 25, 6 (2020), 4547–4584.

[36] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L Glassman. 2020. Enabling data-driven api design with community usage data: A need-finding study. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.