Automated Refactoring to Reactive Programming

Mirko Köhler  
Reactive Programming Technology  
Technische Universität Darmstadt  
Darmstadt, Germany  
koehler@cs.tu-darmstadt.de

Guido Salvaneschi  
Reactive Programming Technology  
Technische Universität Darmstadt  
Darmstadt, Germany  
salvaneschi@cs.tu-darmstadt.de

Abstract—
Reactive programming languages and libraries, such as ReactiveX, have been shown to significantly improve software design and have seen important industrial adoption over the last years. Asynchronous applications—which are notoriously error-prone to implement and to maintain—greatly benefit from reactive programming because they can be defined in a declarative style, which improves code clarity and extensibility.

In this paper, we tackle the problem of refactoring existing software that has been designed with traditional abstractions for asynchronous programming. We propose 2RX, a refactoring approach to automatically convert asynchronous code to reactive programming. Our evaluation on top-starred GitHub projects shows that 2RX is effective with common asynchronous constructs and it can provide a refactoring for 91.7% of their occurrences.

Keywords—refactoring; asynchronous programming; reactive programming; Java;

I. INTRODUCTION

Asynchrony is essential for long-running tasks or computations that involve intensive CPU load or I/O, such as network communication and database access. Hence, asynchronous programming has become more and more important over the last years to allow a large class of software—Web applications (e.g., with AJAX), mobile apps, or Internet-of-Things applications—to stay responsive to user interaction and to environmental input. Also, in a number of cases like event-driven scenarios, where event streams originate from touch screens, GPS, or sensors, asynchrony is the natural programming model as events can be processed concurrently [1].

Most programming languages offer means for developers to define asynchronous computations. A widespread solution is to fork concurrent asynchronous executions using basic thread/process mechanisms, e.g., with `java.lang.Thread` in Java. However, this approach is very low level and complicates the communication with the main thread. For this reason, modern programming languages provide abstractions for asynchronous computations. For example, C# provides the `async` and `await` keywords, and the Java concurrency package has been continuously enhanced through various Java releases to support high level abstractions such as Futures and the Fork/Join execution model [2]. Similarly, popular libraries provide their own constructs for asynchronous computations, such as `AsyncTask` for Android and `SwingWorker` for the Java GUI Swing library. These solutions significantly improve over low level abstractions like threads, but come with their own limitations. For example, `AsyncTask` does not easily support composition, like sequencing multiple asynchronous computations.

Recently, Reactive Programming (RP) has emerged as a programming paradigm specifically addressing software that combines events [3]. Crucially, RP allows to easily express computations on event streams that can be chained and combined using high-order functional operators. This way, each operator can be scheduled independently, providing a convenient model for asynchronous programming. As a result, RP provides means to describe asynchronous programs in a declarative way. Previous research showed that RP improves software design of reactive applications [4], [5]. Also, empirical studies indicate that RP increases the correctness of program comprehension, while requiring neither more time to analyze the program, nor advanced programming skills [6], [7]. This feature has been crucial in popularizing recent RP languages, including ELM [8], Bacon.js and Rx/ReactiveX [9]—originally developed by Microsoft—which received major attention after being adopted for the redesign of Netflix’s streaming server backend.

Unfortunately, most existing software still implements asynchronous computations with traditional techniques. While the benefits of RP design are widely recognized, and new projects can easily adopt RP abstractions right away, this technology is not portable to existing software without a manual refactoring that is tedious and error-prone [10]. As a result, a number of software applications do not benefit from the design improvements provided by RP.

In this paper, we propose an approach to automatically refactoring asynchronous code to the RP style based on ReactiveX’s `Observable`, which is a programming abstraction that emits events that are asynchronously handled by Observers. ReactiveX uses operators from RP to compose computations on Observables. Using these operators, it is straightforward to extend asynchronous computations, thus increasing extensibility. Our methodology applies to common abstractions for defining asynchronous computations in Java, including `Future`, and Swing’s `SwingWorker`. We implemented our approach in 2RX, an extensible Eclipse refactoring plugin. Our evaluation on a number of third-party popular GitHub projects—including Apache Zookeeper, Jabref, JUnit, and Mockito—shows the broad applicability of our technique.
In ReactiveX [9] is a library for asynchronous RP that provides abstractions and operators to process and combine event streams. ReactiveX is available for several languages. In this paper, we consider the Java implementation RxJava. The main abstraction in ReactiveX is an Observable, which is the source of an event stream. Similar to the Observer design pattern, an Observer can register to an Observable and it can be notified of event occurrences. In addition, however, Observables can be chained and executed on different threads. Hence, asynchronous programs can be specified in a pipeline fashion with operators such as map or filter.

Figure 1 demonstrates this design. Initially, we obtain a stream of Data from the network as an Observable (Line 1). Next, all data that is null gets filtered out (Line 3) and the data is transformed to a String and "transformed" is appended (Line 4). The Observable is executed using the default scheduler (Line 5) – ReactiveX provides a multitude of schedulers to enable asynchronous operations. Finally, the result is printed to the command line (Line 7).

This approach has several advantages: the computations over the stream are extensible by adding new operations to the stream and composable via combining streams through functional operators. Since data flow gets disentangled from control flow, programs are easier to comprehend and maintain [4]–[6].

B. Refactoring an OCR Application

Figure 2 shows Juneform [26], an application for Optical Character Recognition (OCR). In Juneform, users load image files in the UI. The application uses asynchronous programming to load the documents and to process them in a separate thread than the UI.

1) Asynchrony in Juneform: The document loading functionality is shown in Figure 3a. The DocumentLoader class, which extends SwingWorker, loads the images into Juneform. The SwingWorker class is provided by the Swing Java GUI library to help programmers implementing asynchronous tasks. In SwingWorker, the asynchronous functionality is defined in the doInBackground method which is executed when the execute method is called. The doInBackground method can both (1) return a value after the computation has finished, and (2) publish intermediate values during its execution.

In the example, DocumentLoader inherits from the SwingWorker class (Line 2) and it assigns two type parameters. The first one is for the return type of the asynchronous call – List<Document> as we want to load a list of documents. The other one is for intermediate results of the asynchronous execution – Document, to start handling documents to the UI thread before the loading of all documents completes. When the user selects the images from
the file chooser dialog and clicks “Open”, the load method 
(Lines 27-30) of DocumentLoader is invoked with the 
selected files as arguments. The files are cached in the private 
files field (Line 28). Finally, execute (Line 29) invokes 
doInBackground (Line 10-22) in a background thread.

Each Document holds the information for an image file, 
e.g., name, absolute path and file content. To load the images, 
the program iterates through all files, creates a Document 
object for each of them and adds it to the results list. Each Document is sent to the UI thread as an intermediate 
result with the publish method (Line 18). Before being 
processed, the results are stored in a buffer (not shown). Then 
the callback method process is invoked with the buffer as 
argument (Line 31). The buffer is necessary as the UI thread may 
not be able to execute process before multiple results have been published. To display the selected images in the UI, 
the process method invokes the fetchResults method 
(Line 40) with the intermediate results of the asynchronous 
operation as argument.

The asynchronous operation ends when the 
doInBackground method returns. The return value of the 
doInBackground method, results, can be 
accessed from the done method using get (Lines 34-38). 
These methods are declared by the SwingWorker class.

2) The Refactoring: The refactoring replaces the 
SwingWorker class with ReactiveX’s Observable 
introduced in Section II-A. There are three aspects to 
consider when refactoring SwingWorker to reactive code: 
(1) Generate emissions on each publish invocation – rather 
than only on the result – to capture the data stream nature 
of SwingWorker, such that an Observer in the RP 
implementation reacts to each emission of the Observable; 
(2) all methods available in the SwingWorker API must be 
available in the Observer and (3) the Observable must 
not stop sending items to the Observer if the latter has not 
finished processing the last emission.

The result of the refactoring is in Figure 3b. The 
getObservable method (Lines 7-25) creates a
DocumentLoader loader = new DocumentLoader();

// Setup observable
Observable<Document> observable = loader.getObservable()
  .flatMap(ev -> Observable.from(ev.getChunks()))
  .filter(doc -> doc.getName().contains(".jpg"))
  .subscribeOn(Schedulers.io());

// Connect observable with subscribers
 observable
  .observeOn(SwooshScheduler.getInstance())
  .subscribe(doc -> uploadToDropbox(document));

// Upload document to server
 observable
  .observeOn(SwingScheduler.getInstance())
  .subscribe(doc -> label.setText(doc.getName()));

// Display document names
 observable
  .observeOn(Schedulers.io())
  .map(doc -> jpgToBitmap(doc))
  .subscribe(doc -> uploadToDropbox(document));

// Connect observable with subscribers
 loader.load(files);

Fig. 4: Extending the Juneiform application.

SWEmitter object that emits a value whenever the doInBackground method produces a result (Lines 9-22). The Observable emits values of type SWEvent. SWEEvent objects carry either an intermediate value or the final result of the asynchronous computation.

SwingWorker has functionalities not available in Observable, such as starting the asynchronous operation with the execute method, or storing the current state of the execution (i.e., whether it has completed). In our approach, the SWSubscriber class implements these functionalities: In the refactored version, DocumentLoader is a subclass of SWSubscriber (Line 2). The load method remains unchanged except calling the executeObservable method (Line 29) to start the asynchronous computation instead of the execute method of the SwingWorker class. Similarly, the process method (Line 31), and the done method (Line 34) override the methods in the SWSubscriber class with the same semantics as the corresponding methods in SwingWorker. The body of all these methods remains unchanged in the refactoring. The user-defined method fetchResults (Line 40) remains unchanged as well.

3) Extending Juneiform: The refactoring described above is of limited interest per se – the approach in Figure 3b is arguably not better than the solution in Figure 3a. However, the refactoring is beneficial because it enables taking advantage of RP’s support for extensibility and asynchronous programming via event stream combination.

Let’s consider the case in which a developer wants to extend the Juneiform application such that (1) only files with jpg extension are loaded, (2) the GUI displays the name of the file currently processed to provide the user feedback on the loading progress, and (3) the image file is converted from jpg to Bitmap and uploaded to Dropbox for backup.

Implementing these changes in the case of Figure 3a is challenging. First, the steps in the computation exhibit dependencies (e.g., transform an image to a Bitmap and then upload it to Dropbox). Hence, a programmer needs to store intermediate results – hiding the logic of the processing sequence. A harder problem is that the long-running tasks in (3) should be executed in separate threads and processing should happen as soon as new files are available. Finally, (2) should be executed on the GUI thread to avoid race conditions on the (non-synchronized) GUI components.

These functionalities can be of course achieved with a complex combination of non-composable imperative transformations, intermediate state, asynchronous constructs for the execution on separate threads, and callbacks to achieve responsive push-based processing of new files.

Alternatively, developers can implement these features right away with RP operators such as filter. Figure 4 shows an implementation of these features that extends the refactored version from Figure 3b. First, the getObservable method retrieves the Observable (Line 5) previously defined in the DocumentLoader class. For (1) – load only jpg files – the flatMap operator applied to the Observable (Line 6) provides a stream of Documents, since the Observable produces events containing values of type List<Document>. From this stream, the filter method produces a stream with only jpg files (Line 7). These functionalities are executed in a background thread as specified by Schedulers.computation() (Line 8). For (2) – display in the GUI the name of the file currently processed – Lines 11-13 directly specify to execute this operation in the UI thread. Finally, for (3), the file is converted from jpg to Bitmap as well as uploaded to Dropbox (Lines 16-19). These operations are executed asynchronously and independently of each other – the latter executes as soon as a result from the former is available. Loading the files starts the emission (Line 22). Further extensions are also trivial to support and can be implemented adding new operators to the pipeline.

4) Discussion: We have shown that, thanks to the refactoring to RP, the asynchronous Juneiform application benefits from the advantages of RP design – extensibility and compositability. The refactoring we just described requires some observations. In the example, the refactored code is slightly longer and it introduces new concepts, such as Observable or Subscriber, which may harden program comprehension for developers. However, there is no way around learning new concepts when switching to a new programming paradigm, and our target are developers who are already familiar with RP. On the other hand, automated refactoring may help novices to adapt to RP without the burden to manually introduce the new concepts in the codebase. Another potential downside is that the refactored code may not be as performant. For example, one factor that influences performance is the configuration of the scheduler for the asynchronous computations (e.g., Futures), which cannot be faithfully reproduced in the refactored code. Yet, RxJava provides access to the scheduler configuration so that the developer can easily tune the scheduling policy, e.g., by subscribing an Observable to a different scheduler.
III. 2Rx: Refactoring to Reactive Programming

We designed 2Rx, a tool that automatically refactors selected asynchronous constructs to ReactiveX. The key idea of the refactoring is to transform the values generated by asynchronous computations into an event stream that emits an event whenever a new value is generated. For example, SwingWorker produces (several) intermediate values and a final result, both of which can be represented as an event stream. 2Rx’s refactoring strategy is to replace the asynchronous construct with a ReactiveX Observable. The Observable emits an event for each results generated by the asynchronous computation. We target two Java constructs for asynchronous programming:

- javax.swing.SwingWorker: SwingWorker is a construct defined in the Java Swing library. It asynchronously executes the code in its doInBackground method, which returns the result of the computation. Also, SwingWorker can emit intermediate values during the asynchronous execution.
- java.util.concurrent.Future: A future is a value that has not been computed yet, but is available eventually [27]. In Java, the Future interface has several implementations. Although Future does not emit multiple values, refactoring to RP relieves developers from handling the emission of the value explicitly and enables RP’s support for functional composition and asynchronous execution.\(^1\)

The constructs above are not the only way to define an asynchronous task. For example, a primitive way to implement asynchronous execution of long-running computations is via threads, i.e., by implementing the java.lang.Runnable interface or by subclassing the java.lang.Thread class.

2Rx does not support refactoring threads, because threads are not only used for asynchronous computations, but also for, e.g., parallel execution and scheduling. Threads expose a general interface to programmers which enables executing any computation in the thread. The communication with other threads takes place through shared state. Crucially, threads do not provide a well-defined interface for asynchronous computations that return values, making them unsuitable for refactoring to RP as there is no return value that can be processed by the Observable chain. Unfortunately, the cases in which threads express proper asynchronous computations are hard to distinguish because assignment to shared state can equally be due to thread communication in a concurrent application or to synchronization among parallel tasks – without the programmer intention being explicit.

Many modern libraries/languages provide dedicated support for asynchronous computations, like C#’s async/await, Java’s java.util.concurrent.Future, and Scala’s Async, substantially deprecating the use of raw threads for concurrency. Our approach targets these cases.

\(^1\)For this reason, RXJava already provides a fromFuture method to convert a Future into an Observable.

For the java.util.concurrent.ExecutorService class, considerations similar to those made above for Threads apply. The class provides an API to submit asynchronous computations and to decouple task submission from the mechanics of running the task, i.e., thread use and scheduling. Yet, in some cases, the Executors class produces Java Futures. In the cases where the return value is used by the rest of the program, hence the future object is available, 2Rx performs the refactoring for java.util.concurrent.Future.

A. Refactoring Asynchronous Constructs

In this section, we provide an overview of our refactoring approach for each asynchronous construct.

1) SwingWorker: Refactoring SwingWorker to Observable requires to consider two major differences between the two constructs: (1) SwingWorker does not only emit a final result, but also intermediate results with a different type, and (2) SwingWorker keeps track of the current status of the computation – if it is still running or if it has already finished. To achieve the functionalities of SwingWorker with Observable, we implemented an extension to ReactiveX consisting of three helper classes (Figure 3b): SWEmitter emits an event for each call to the publish method and for the final result (Line 8); SWSSubscriber implements the SwingWorker API on top of the emitter (Line 2); SWEvent is the type of events produced by the Observable, holding either an intermediate or a final result (Line 7). These three classes are added to the project’s class path during the refactoring.

The actual refactoring consists of replacing SwingWorker with SWSSubscriber. The SwingWorker definition is split into two parts by 2Rx. (1) The Observable is wrapped around the SWEmitter class to support further asynchronous computations. (2) The SWSSubscriber class subscribes to the Observable and provides the SwingWorker API. The body of the SwingWorker class is then rewritten such that the doInBackground method is placed in the SWEmitter. The other methods remain unchanged.

2) Java Futures: Figure 5a shows the use of a Future in conjunction with an ExecutorService to schedule asynchronous tasks. The example is taken from Elasticsearch [28], a distributed search engine.

First, the ExecutorService is initialized (Line 1), and a list of Future is created (Line 3). A task is submitted to the ExecutorService (Line 7). The task is defined in the call method of Callable (Lines 9-15) which returns a Future for the task result that is then added to the list. In the example, the task waits for the result of another executor and retrieves the result – the List\(<\text{T}>\) returned by the Future (Line 14). The task is executed asynchronously, i.e., it does not block the execution of the main thread. The computation only blocks when the result of the Future is retrieved to store it in a list of results (Line 20).

Figure 5b shows the refactored version. The initialization of the ExecutorService does not change (Line 1). The list now stores Observable instead of Future (Line 3). An
Observable is created from the same Callable that was submitted to the executor (Lines 9-15). The Observable uses a Scheduler (Line 16) to run asynchronously. Note that the Future is still executed according to the ExecutorService – only the Observable operates on the Scheduler.

3) Discussion: The drawback of the helper classes used in the SwingWorker refactoring is that they are an extension to the ReactiveX library, while a clean refactoring should be based on ReactiveX only. Helper classes enable refactoring more cases, as they take over some of the responsibility of preserving the functionality during the refactoring. But, helper classes also complicate the code introducing additional classes and functionalities. We believe that the satisfactory refactoring results for Java Future do not justify the loss of such clean approach for those cases.

B. Checking Preconditions

Before applying a refactoring, it is crucial to check whether certain conditions (refactoring precondition [29]) are satisfied to guarantee that the refactoring is possible and the result is correct. Figure 6 shows a problematic case. The original code is in Figure 6a. Two futures are created (Line 1 and 4). The first future does not use the standard Future interface but the Future subclass ScheduledFuture, on which the cancel method is invoked (Line 9). This configuration leads to a non-refactorable Future instance, because (1) Observable does not support the equivalent of a cancel method in the Future class, and (2) the ScheduledFuture subclass provides additional functionalities that are not supported by a refactoring that targets Future. The precondition check finds non-refactorable Future instances and excludes them from the refactoring.

The refactored code is in Figure 6b. Because of the problem above, the first Future is not refactored (Line 1) and our tool issues a warning for it. The second Future instance passes the precondition check as, contrary to the first Future, it uses the default Future interface and it only calls the get method. The refactoring for this instance is correct (Line 4 and 7). The example demonstrates the need for flow-sensitive source code analysis in 2R, e.g., to find out on which instances cancel is invoked, and to selectively reject refactoring those instances.

Our preconditions are defined to disallow using functionalities of asynchronous constructs that can not be translated into corresponding functionalities of Observable. We define similar preconditions as Yin and Dig [30]: Asynchronous constructs are only used by executing and retrieving their result, and subclasses are disallowed. Preconditions are defined on occurrences of asynchronous constructs. An occurrence is the coherent usage of a construct. For example, in Figure 6, there are two occurrences of Future. Concretely, we define three preconditions for our refactoring approach. Occurrences where at least one precondition is not satisfied are excluded from the refactoring.

- The asynchronous computation is not cancelled. ReactiveX provides no way to cancel asynchronous computations of Observables, but only to unsubscribe an observer (which does not cancel the running computation).
- The state of an asynchronous execution is not retrieved. Asynchronous constructs often provide ways to directly retrieve the current state of an asynchronous execution, e.g., Future.isDone() checks whether an asynchronous computation has completed. Observables lack such functionality.
- The asynchronous construct is not used through a subclass that provides additional functionalities. This case includes classes specific to a single project, as well as unsupported library constructs like, e.g., ScheduledFuture from Figure 6.
C. Implementation

We designed 2Rx as an Eclipse plugin for refactoring Java projects. The plugin consists of an extensible core that provides Eclipse integration and an API for the basic functionalities required by all refactorings. The API allows retrieving the AST of a compilation unit, performing static data flow analyses, identifying a specific Java construct, manipulating the AST, and outputting the refactored code. We implemented an extension for the two asynchronous constructs in Section III, AST, and outputting the refactored code. We implemented an analyses, identifying a specific Java construct, manipulating the AST, and outputting the refactored code. We implemented an extension for the two asynchronous constructs in Section III, Java Future and SwingWorker. Each extension uses the API to search for a specific construct and to implement the associated transformation. We have implemented an automatic precondition checker for both the constructs currently supported. The checker is based on a flow-sensitive static analysis on Java source code [31]. The extensible architecture simplifies adding new refactorings for other constructs, which is ongoing work.

IV. Evaluation

In this section, we evaluate our refactoring approach. First, we define our research questions.

A. Research Questions

Our evaluation answers the following research questions. First, we want to evaluate which fraction of all asynchronous constructs available in Java can be targeted by 2Rx.

Q1 (Applicability): Which fraction of asynchronous constructs used in real-world projects is supported by 2Rx?

Second we want to know, among the occurrences of asynchronous constructs supported by 2Rx, how many cases satisfy the preconditions that 2Rx requires to perform the refactoring and lead to refactored code that is correct and achieves the same functionality of the original code.

Q2 (Correctness): How many occurrences of the supported asynchronous constructs can 2Rx correctly refactor?

Finally, we want to ensure that our refactoring tool is usable, e.g., the refactoring process must complete in a reasonable time even for large projects.

Q3 (Efficiency): Is 2Rx fast enough to be usable by developers?

Next we present the methodology for creating the datasets to evaluate our work. Finally, we present the answer to the research questions above, e.g., the evaluation results on applicability, correctness and efficiency of our solution.

B. Methodology

In the following, we describe the methodology to answer the research questions.

1) Global Dataset: First, we identified the asynchronous constructs that are used in Java. We looked for documentation and tutorials that introduce asynchronous programming in Java and we considered the documentation of frameworks, such as Spring [32], which provide asynchronous constructs to users. Further, we talked to other researchers and developers, personally and over StackOverflow, about which asynchronous constructs they use. With this process, we have identified the 14 asynchronous constructs listed in Table I. We used Boa [33] – a language and infrastructure for mining software repositories – to identify projects that contain these constructs. Boa provides a snapshot of all public GitHub projects from 2015, consisting of 380,125 projects. We considered the 275,879 projects containing Java source files. We found 46,208 Java projects that contain at least one asynchronous construct.

The global dataset is composed of GitHub open-source projects that use one of the constructs above. We identified projects containing an asynchronous construct by searching for the class name and the corresponding import in the AST. We also searched for classes that are created by the factory classes akka.dispatch.Futures and java.util.concurrent.Executors, such as Future, Executor, ExecutorService and ScheduledExecutorService, respectively. For the class javax.enterprise.concurrent.ManagedTask, we also considered projects that use one of the associated classes ManagedExecutorService or ManagedScheduledExecutorService. This approach ensures that we find constructs that are adopted in the project even if they do not appear syntactically in the code (e.g., in case they are returned by a method call and used without assigning them to an intermediate variable). We used the global dataset to evaluate applicability of our methodology.

2) Refactoring Dataset: We filtered the global dataset to select relevant projects for each of the two constructs supported by 2Rx, generating the refactoring dataset.
First, we filtered out all projects that do not contain one of the two constructs supported by 2Rx: SwingWorker, and Java Future. Out of the 46,208 Java projects in the global dataset, we found a total of 7,133 projects that use at least one of the two supported constructs; 5,718 projects use Java Future and 1,651 projects use SwingWorker (236 projects use both). Next, we removed projects that do not use one of the popular build tools Maven [34] or Ant [35] to – at least partially – automate the evaluation, as we automatically tested the refactored code as well as checked whether the refactored code compiles. We identified projects that use a build tool based on the respective build file, i.e., pom.xml for Maven and build.xml for Ant. We ended up with 4,652 projects for Future and 1,118 projects for SwingWorker.

On GitHub, projects can be starred by other users to follow their updates. We used GitHub stars as a metric for the popularity of a project [36]. As stars are not included in their updates, we used GitHub stars as a metric for the popularity of a project. As we were able to compile, resulting in the refactoring dataset. As we use the Boa 2015 dataset, some projects required dependencies that were no longer available. We filtered out these projects as we were not able to compile them. For Future, this is the case for 5 projects, for SwingWorker, it is the case for 16 projects. The datasets that we created are available to researchers interested in studying Java asynchronous constructs.

3) Automatic Test Generation: To evaluate the correctness of the refactorings, we ran unit tests for each project. We assume that, when the unit tests pass before as well as after the refactoring, the refactoring does not change the functionality of the code. Unfortunately, for many projects, existing tests provided in the project already failed even with the original code, or did not cover the refactored code. For this reason, we opted for automatic test generation to create tests for exactly the parts of the code that were refactored.

To automatically test the refactored code, we implemented a framework based on Randoop [37]. The generated tests capture the behavior of the original code, and are run again on the refactored code to ascertain that programs are behaviorally equivalent to the original code. Automatic test generation to assess correct refactorings has been used before, e.g., by Soares et al. [38]. Formal verification of the correctness of refactorings is a very recent research direction [16].

To ensure that test generation remains feasible for large projects, the testing framework accurately generates tests only for methods changed by the refactoring. When the refactoring changes the signature of a method, i.e., name, parameter or return types, the framework generates tests for methods that call the changed method. The same holds for methods that are not directly accessible, e.g., because they are private. Methods that are nondeterministic or have side-effects across method calls (such as writing/reading from a file) are not amenable for automated testing, because calls to the same method result in different behaviors. The framework detects nondeterministic methods by recognizing varying test results for those methods. All tests for nondeterministic methods are discarded.

4) Evaluation Methodology: For Q1 (Applicability), we determined the most common asynchronous constructs used by developers in the global dataset and reported which percentage of them is supported by 2Rx. For Q2 (Correctness), we checked which fraction of the occurrences of asynchronous constructs in the refactoring dataset are correctly refactored by 2Rx. Correctness is evaluated by checking that the project still compiles after the refactoring and by executing the generated tests. For compilation and testing, we use the Boa 2015 version of the projects. We only considered occurrences that 2Rx actually refactors, i.e., occurrences that are not rejected by the preconditions. For Q3 (Efficiency), we measured the time that 2Rx requires to refactor a project. This time includes parsing the source code and constructing the AST, finding the appropriate constructs, analyzing them, and generating the refactored code.

C. Results

Table I and Table II summarize the evaluation results from, respectively, the global dataset and from the refactoring dataset. We describe the tables and then we refer to them to answer our research questions.

In Table I, for each construct, we show the total and relative number of projects where the construct is used. We found a total of 46,208 projects that use at least one asynchronous construct. Table II provides the results of the refactoring for each asynchronous construct in the refactoring dataset.

a) Q1 (Applicability): Table I shows that Thread and Executor are the two most used asynchronous constructs in Java. Similar to Thread, the Executor interface can be used for arbitrary concurrent computations, usually via its ExecutorService extension. However, contrary to Thread, Executor provides an API to submit

<table>
<thead>
<tr>
<th>Construct</th>
<th>Total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Thread</td>
<td>40,988</td>
<td>14.86</td>
</tr>
<tr>
<td>java.util.concurrent.Executors</td>
<td>11,708</td>
<td>4.24</td>
</tr>
<tr>
<td>java.util.concurrent.Future</td>
<td>5,718</td>
<td>2.07</td>
</tr>
<tr>
<td>java.util.concurrent.SwingWorker</td>
<td>1,651</td>
<td>0.00</td>
</tr>
<tr>
<td>java.util.concurrent.FutureTask</td>
<td>1,372</td>
<td>0.50</td>
</tr>
<tr>
<td>com.google....ListenableFuture</td>
<td>295</td>
<td>0.11</td>
</tr>
<tr>
<td>java.concurrent.Thread</td>
<td>163</td>
<td>0.06</td>
</tr>
<tr>
<td>akka.dispatch.Futures</td>
<td>122</td>
<td>0.04</td>
</tr>
<tr>
<td>java.util.concurrent.ForkJoinTask</td>
<td>92</td>
<td>0.03</td>
</tr>
<tr>
<td>java.util.concurrent.AsyncResponse</td>
<td>82</td>
<td>0.03</td>
</tr>
<tr>
<td>javax.ws.rs.container.AsyncResponse</td>
<td>31</td>
<td>0.01</td>
</tr>
<tr>
<td>javax.enterprise.concurrent.ManagedTask</td>
<td>19</td>
<td>0.01</td>
</tr>
<tr>
<td>java.util.concurrent.CompletableFuture</td>
<td>7</td>
<td>&lt;0.01</td>
</tr>
<tr>
<td>org.springframework.scheduling.annotation.Async</td>
<td>1</td>
<td>&lt;0.01</td>
</tr>
</tbody>
</table>

TABLE I: Asynchronous constructs in Java – global dataset.

For each asynchronous construct, the absolute number of projects (Column total) and the percentage over all projects that use at least one asynchronous construct (Column %) is given. Projects can contain multiple asynchronous constructs.
TABLE II: Refactoring popular GitHub projects — refactoring dataset.

Column Stars shows the number of stars (i.e., “followers”) on GitHub. Column cond shows how many occurrences pass (y) or violate (n) the preconditions, respectively. Column time indicates how long it takes to refactor the whole project. LOC files are the lines of code, and files the number of source files in the pre-refactored project both measured with CLOC [39]. Columns c? and t? indicate whether the refactored program has been successfully compiled or tested, respectively. ✓ indicates that the compilation/testing was successful, and ✗ in the t? column indicates that the automatic test generation failed.

<table>
<thead>
<tr>
<th>Stars</th>
<th>Project</th>
<th>cond</th>
<th>Time (ms)</th>
<th>LOC</th>
<th>files</th>
<th>c?</th>
<th>t?</th>
</tr>
</thead>
<tbody>
<tr>
<td>3058</td>
<td>Zookeeper</td>
<td>y</td>
<td>16,051</td>
<td>83,956</td>
<td>671</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>1777</td>
<td>Disunity</td>
<td>n</td>
<td>7,733</td>
<td>5,720</td>
<td>113</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1285</td>
<td>Gitbit</td>
<td>y</td>
<td>9,926</td>
<td>32,244</td>
<td>283</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>661</td>
<td>OptaPlanner</td>
<td>y</td>
<td>20,756</td>
<td>93,268</td>
<td>698</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>565</td>
<td>Jabref</td>
<td>y</td>
<td>31,138</td>
<td>60,219</td>
<td>946</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>486</td>
<td>Nodebox</td>
<td>y</td>
<td>13,676</td>
<td>73,597</td>
<td>732</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>150</td>
<td>ATLAncher</td>
<td>n</td>
<td>5,941</td>
<td>46,292</td>
<td>348</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>109</td>
<td>CookieCadger</td>
<td>y</td>
<td>1,056</td>
<td>10,217</td>
<td>132</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>89</td>
<td>PIPE</td>
<td>y</td>
<td>13,676</td>
<td>73,597</td>
<td>732</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>70</td>
<td>BurpSentinel</td>
<td>y</td>
<td>1,056</td>
<td>10,217</td>
<td>132</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(a) SwingWorker.

<table>
<thead>
<tr>
<th>Stars</th>
<th>Project</th>
<th>cond</th>
<th>Time (ms)</th>
<th>LOC</th>
<th>files</th>
<th>c?</th>
<th>t?</th>
</tr>
</thead>
<tbody>
<tr>
<td>23495</td>
<td>Elasticsearch</td>
<td>y</td>
<td>261,132</td>
<td>370,006</td>
<td>3,595</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6152</td>
<td>JUnit</td>
<td>y</td>
<td>4,577</td>
<td>24,218</td>
<td>375</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5820</td>
<td>DropWizard</td>
<td>y</td>
<td>48,910</td>
<td>17,708</td>
<td>361</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4871</td>
<td>Mockito</td>
<td>y</td>
<td>104,409</td>
<td>52,871</td>
<td>822</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4790</td>
<td>Springside4</td>
<td>n</td>
<td>8,266</td>
<td>20,293</td>
<td>199</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>4424</td>
<td>Titan</td>
<td>y</td>
<td>116,117</td>
<td>40,301</td>
<td>531</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>3774</td>
<td>AsynclntClient</td>
<td>y</td>
<td>15,402</td>
<td>29,739</td>
<td>344</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>3327</td>
<td>Graylog2Server</td>
<td>y</td>
<td>47,839</td>
<td>138,663</td>
<td>2,014</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3018</td>
<td>Java Websocket</td>
<td>y</td>
<td>1,653</td>
<td>5,117</td>
<td>52</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2840</td>
<td>B3log</td>
<td>y</td>
<td>16,217</td>
<td>14,635</td>
<td>173</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) Java Future.

asynchronous computations which return a Java Future. We are able to refactor these occurrences of Future with 2Rx. These occurrences are already included in the projects using Futures in Table I. Java Future, i.e., the java.util.concurrent.Future interface, provided in the Java standard library, is the next most used construct. Standard ways to create a Future are either through ExecutorService – as we already mentioned – or by directly instantiating one of the various Future implementations: FutureTask, SwingWorker and AsyncResult implement the Future interface. The other constructs are responsible for the remaining asynchronous computations.

2Rx provides refactoring support for SwingWorker, and java.util.concurrent.Future as well as FutureTask. FutureTask is an implementation of the Future interface that does not add any further functionality. For this reason, 2Rx is able to refactor this construct. As already discussed (Section III), Thread and Executor are excluded from the refactoring because they express computations that are not suitable for the RP model. As a result, 2Rx targets a total of 7,552 projects, which is ∼2.7% of all projects, and ∼16.3% of all projects that contain at least one asynchronous construct. We conclude that 2Rx is applicable to some of the most used Java constructs for asynchronous computations.

b) Q2 (Correctness): Table II presents the refactoring dataset with the amount of asynchronous constructs refactored for each project and the results of the automatic testing. As described in Section III-B, the code needs to fulfill refactoring preconditions to ensure that the refactoring can be applied correctly. We evaluate the correctness of the refactoring by checking that the refactored code (1) can be compiled (ensuring that the refactoring produces valid Java), and (2) passes the automatically generated tests.

In the SwingWorker case (Table IIa), all occurrences of asynchronous constructs pass the preconditions. Compilation after refactoring succeeds for every project. Automatic test generation fails for 3 projects. The reason is that the methods under test are within inner classes, hence it is not possible to automatically generate tests with Randoop (Section IV-B3). For the other projects, all tests succeed.

In the case of Future (Table IIb), we were able to refactor ∼89.8% of the occurrences. As mentioned in Section III-B, the precondition check excludes asynchronous constructs that are (1) subclasses of Future, that are (2) cancelled via Future.cancel(), or that (3) retrieve the state of the asynchronous computation via Future.isCancelled() or Future.isDone(). The precondition check rejects 13 occurrences, where 6 are cancelled, 6 are subclasses, and 1 is both. The compilation of all projects succeeds after refactoring, but the automatic test generation fails for 3 projects, because of inaccessible methods and abstract classes. For the other 7 projects, tests are generated successfully and the tests succeed.

The evaluation shows that the use of helper classes in the case of SwingWorker allows us to perform significantly more refactorings than in the case of Future for which no helper classes are used. In summary, 2Rx is capable of refactoring ∼91.7% of cases. All refactored cases can be compiled correctly. For 6 projects, the automatic test generation failed, but for the other projects, all tests are executed successfully.

c) Q3 (Efficiency): At last, we discuss the efficiency of our approach. We present the run time for each refactoring in Table II. The times were measured on an Intel Core i5-7200U, with 16 GB memory running Linux Kernel 4.15.

On average, our approach requires ∼366 ms/1K LOC for SwingWorker, and ∼1121 ms/1K LOC for Java Future. The SwingWorker refactoring is faster than the Future refactoring, because the precondition analysis of Future is more involved, since the helper classes in the SwingWorker refactoring obviate the need of preconditions. The upper bound of the refactoring time, the Elasticsearch project in the Java Future refactorings, requires ∼261 seconds. This relatively high amount of time is due to the size and the complexity of the code base. The Elasticsearch project consists of a total of 3,602 compilation units, and 261,132 LOC. We still consider this time acceptable, as the refactoring only has to be done once for the entire project: It is faster than the time that
would be required for a programmer to find the 4 asynchronous
constructs and to manually refactor them. For smaller projects,
the refactoring is faster and only takes a few seconds.

V. THREATS TO VALIDITY

In this section, we discuss the threats to the validity [40] of
our work.

1) Internal Validity: The results of the evaluation of cor-
rectness rely on our testing framework described in Sec-
tion IV-B3. The framework generates extensive tests that
increase the confidence in the behavioral equivalence of the
original code and of the refactored code. Testing is, how-
ever, by definition not complete and it could be that some
differences are not detected. To mitigate this threat, we also
inspected all refactorings manually and found them correct.
The manual inspection has been done independently by two
scientific staff members not involved in the project and by
the authors. When necessary, we investigated conflicts in
the inspection by manually writing small code examples which
capture the essence of the inspected code. We then tested that,
after refactoring, the examples behave the same as before. It
is possible that refactored methods cannot be tested directly,
because, e.g., methods are not accessible or the refactoring
changes a methods signature. For this reason, the test gener-
ation framework creates tests for methods that call the refac-
tored methods. This approach increases the number of methods
that can be tested. Some refactoring cases, however, cannot
be automatically tested, because they rely on nondeterministic
methods. In these cases, the evaluation of the refactoring has
to rely on manual inspection and on ensuring that the project
compiles correctly. Nondeterministic methods are detected by
the testing framework and are excluded from the automatic
test generation. In summary, we use the automated testing
approach to increase our confidence in the soundness of our
work, rather than relying on it. There exist ad hoc testing
techniques that specifically target certain refactorings [41], but
none of them directly apply to the refactorings used in this
paper, opening an interesting direction for future work.

2) External Validity: Another threat to validity is whether
our results can be generalized. The projects considered for
the evaluation are open-source projects available on GitHub.
We considered projects that have a high amount of stars
(followers) which is on one hand an indication of popular-
ity, but also of a certain code quality [36]. The refactoring
dataset contains software applications that belong to different
domains, including a distributed database (Titan), a computer
game launcher (ATLauncher) and a testing framework (JUnit).
In addition, since we adopt projects in the wild for our
evaluation, we consider codebases developed by different
teams, which promises variety in coding style. In particular,
manual inspection showed a diverse usage of asynchronous
constructs, e.g., Future is used with Executor, custom
implementations, or as part of collections, amongst others.

These considerations increase our confidence that the results
presented in this paper generalize to most Java projects that
include the supported asynchronous constructs.

VI. RELATED WORK

We organize related work into languages for functional
reactive programming, recent results on refactoring techniques,
and approaches to refactor existing software to asynchronous
execution.

1) Reactive Programming: Functional reactive program-
ing has been introduced in the Fran [42] animation system.
Over the last years, abstractions from FRP – such as event
streams, time-changing values and combinators to support
their modular composition – have been introduced in a number
of extensions of mainstream languages. These include Ba-
(Scheme), REScala [44] (Scala), Scala.React [45] (Scala),
and RxJava [9] (Java). Such approaches differ, among the
other features, in the abstractions they support (e.g., event
streams and behaviors [4], [5], [44], or events only [9], [43])
as well as in the consistency guarantees they provide (e.g.,
 glitch-freedom [4], [5], [44], eventual consistency [9], [43] or
multiple consistency levels [46]). Some approaches (e.g., [44])
use synchronous RP. For a complete overview of RP the reader
can refer to the survey by Bainomugisha et al. [3].

Tool support for reactive programming is still in an early
stage, with recent proposals to support interactive debug-
ning [47], [48].

2) Refactoring: Refactoring refers to the process of mod-
ifying a program – usually to improve the quality of the
code – such that the behavior remains the same [29], [49].
In a broader sense, refactoring can also change nonfunctional
requirements of programs, e.g., performance. Studies have
shown that refactoring can improve maintainability [50] and
reusability [51] of existing software. Unfortunately, despite
the availability of dedicated tools, developers tend to apply
changes manually to fulfill the new requirements [52].

Operationally, refactoring consists of a series of transfor-
mations after checking that the refactoring candidate satisfies
some preconditions. If any step in the sequence fails, all the
transformations already done should be rolled back [53].

Due to space reasons, it is not possible to provide a
complete overview of previous research on refactoring. We
only focus on recent work in representative areas. Wloka et
al. propose a refactoring to transform global state of Java
programs to thread-local state, where possible, to introduce
parallelism [17]. Schäfer et al. [18] propose a refactoring
from Java’s built-in locks to a lock implementation that can
be fine-tuned to improve performance. Schäfer et al. [54]
found that many refactorings are not safe for concurrent code
and propose a technique to make refactoring concurrent Java
source code more reliable. Tsantalis et al. propose to use Java 8
lambda expressions to refactor clones with (small) behavioral
differences [19].

Other recent refactoring approaches target introducing
Java 8 default methods into legacy software [20], automatic
detection of refactorings for object-oriented models [21], clone
refactoring [22], as well as performance improvement [23]
using a database with an ad hoc representation of source
code. Liebig et al. [55] address refactoring with preprocessor directives and propose an approach to preserve the behavior of all variants. Meng at al. propose to use refactoring for clone removal [56]. Jongwook et al. investigate scripting refactorings to retrofit design patterns in Java [57]. LAMBDAFICATOR [14] is a tool that – motivated by the new features of Java 8 – converts anonymous inner classes to lambda expressions and converts for-loops that iterate over Collections to high-order functions. None of these works tackles the issue of refactoring existing applications to RP.

3) Refactoring Asynchronous Code: The line of work by Dig and colleagues, addressed the issue of refactoring asynchronous code in the context of mobile applications. This line of research starts from the observation that most available documentation focuses on creating asynchronous apps from scratch rather than on refactoring existing synchronous code. Hence not enough tools are available to support developers to perform the refactoring [1].

With Asynchronizer [11], developers select certain statements of a program and the tool creates an AsyncTask which executes the selected code asynchronously. A static analysis informs the programmer in case the generated code introduces data races.

Lin et al. [24] found that many developers use the wrong construct when dealing with the GUI: in Android, the asynchronous construct IntentService is more efficient than AsyncTask which uses shared-memory. Lin et al. developed ASYNCDROID, a tool for refactoring AsyncTask into IntentService.

Okur et al. [12], propose a tool to refactor asynchronous C# programs based on two components. ASYNCFIYER transforms legacy code that uses callbacks into code that uses the built-in async/await constructs. CORRECTOR recognizes performance anti-patterns and issues like deadlocks of async/await constructs. The tool corrects most cases of undesirable behavior found through an empirical study based on GitHub public repositories.

Brodu at al. [25], propose a tool to transform callbacks to Dues – a construct similar to Promises [27], [58] – based on the observation that callbacks lead to code that is hard to understand – an issue known as Callback Hell [59].

All these approaches either (1) introduce asynchronous execution for otherwise sequential code [11], or (2) refactor asynchronous code to use more specific constructs (e.g., async/await [12], IntentService [24], Promises [25]). Our approach is complementary to (1) as it can refactor applications to RP after asynchronous execution has been introduced. Conceptually, our approach targets the output of (1) and (2) to convert it to RP.

VII. CONCLUSION

In this paper, we presented a method to refactor asynchronous programming constructs to RP. We implemented an Eclipse plugin which automates the refactorings and we evaluated it on popular third-party projects from GitHub. The results show that our tool correctly refactors some common constructs for asynchronous programming in Java. We are currently extending 2Rx to support more constructs and improve its applicability. We hope that, equipped with 2Rx, more and more programmers can bring the design benefits of RP to their projects.

ACKNOWLEDGMENT

This work has been performed in the context of the LOEWE centre emergenCITY and it has been supported by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI and 1119 CROSSING, by the DFG projects 383964710 and 322196540, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the AWS Cloud Credits for Research program.

REFERENCES
