From Kotlin to Swift and Back: Toward Fully Automated Cross-Language Code Transpilation

Sachi Lad
University College London, UK
sachi.lad.21@alumni.ucl.ac.uk

Carol Hanna
University College London, UK
ORCID: 0009-0009-7386-1622

Justyna Petke
University College London, UK
ORCID: 0000-0002-7833-6044

***Abstract*—Mobile platforms dominate software development, yet transpiling code between Kotlin (Android) and Swift (iOS) remains a major challenge. This task is essential for cross-platform accessibility, particularly when porting iOS apps to Android. Despite its importance, research on Kotlin–Swift transpilation and particularly, fixing these transpilation bugs is scarce. We thus present the first large-scale study of Kotlin–Swift transpilation bugs. We identify 149 real-world bugs, introduce a taxonomy of common bug types, and propose new mutation operators to address them. Combined with existing operators, our proposed operators have potential to fix 101 of these bugs. Building on the taxonomy of transpilation bugs and the mutation operators we develop to repair them, we lay out a research agenda for reliable, fully-automated, end-to-end software transpilation workflows for the mobile domain.**

***Index Terms*—mobile, transpilation, bug repair, Swift, Kotlin**

I. Introduction

Mobile applications are ubiquitous, with the industry dominated by Android and iOS [1]. As such, developers often need to transpile code between Kotlin (Android) and Swift (iOS). Transpilation, automatically transpiling source code written in one programming language to another [2], can streamline this process. Reliable automated transpilation between Kotlin and Swift would improve developer productivity, reduce duplication, and enhance accessibility. This is especially important when porting applications from iOS (Apple) to Android as some applications are developed solely for iOS, despite Android being more widely used due to affordability [3].

Despite this practical need, research on Kotlin-Swift transpilation is limited, as evidenced by the small number of mobile-focused papers identified in a previous systematic review [4]. Schneider and Schultes conducted an evaluation of three rule-based transpilers for Kotlin and Swift [5] that analysed transpilation issues. However, the study did not provide a bug taxonomy nor offer solutions to fix the issues, as the focus was on language coverage of the subject transpilers. Additionally, the results may not reflect the current state of the art, due to updates to transpilers since. There is existing work on fixing transpilation bugs in general [6] but none focus on mobile-specific languages like Kotlin and Swift. Recent research [6] shows that transpilation remains an unsolved problem, even in well-studied classical language pairs. For mobile, the challenge is even more underexplored, despite high practical impact.

In this study, we situate our approach within the context of Automated Program Repair (APR) [7], [8], in particular, the search-based branch. It uses heuristics to explore the vast search space of possible changes to the code via *mutation operators* to automatically find fixes for software bugs. In this work, first, we transpile a set of input code based on Kotlin and Swift language constructs bi-directionally, using a rule-based and an LLM-based transpiler. We then manually analyse the transpiled code to identify and classify bugs into an appropriate bug taxonomy. Following this, we suggest how existing mutation operators could be used to fix bugs in an automated manner. We propose 3 novel mutation operators specifically targeted at fixing transpilation bugs between Kotlin and Swift. Overall, we identified 149 bugs across all transpilations, which we classified into an existing bug taxonomy, and further classified the bugs into derived finer-grained categories. Finally, we test the efficacy of large language models in fixing the identified transpilation bugs. This study is a step forward towards fully automated Kotlin-Swift transpilation for mobile, and highlights promising directions for future research. Insights from our study can help developers identify transpilation bug patterns in mobile apps, provide fix patterns for search-based APR tooling, and as context for LLM-based approaches. To aid replicability we provide our data [9].

II. Kotlin-Swift Transpilation Study

In order to understand the types of errors in mobile code transpilations and propose ways of automating their repair, we conducted a study guided by the following questions: **RQ1:** What is the effectiveness of the selected tools in code transpilation between Kotlin and Swift? **RQ2:** What types of bugs occur when transpiling between Kotlin and Swift? **RQ3:** What types of mutation operators can be used to reduce the number of bugs in the transpiled code? **RQ4:** What is the efficacy of LLMs guided by fix patterns in improving the accuracy of bug fixing in Kotlin-Swift code transpilation?

1) Transpilers: SequalsK [10] was chosen as the subject transpiler out of available (7 total) tools, as it was the only one that was bi-directional, not deprecated, and showed promising results in previous work [5]. CodeLlama [11] was chosen as the subject LLM following a review of relevant literature. It meets two key criteria: it has empirical evidence of being an effective transpilation tool [12] [13] and it is open-source, providing wide availability.

2) Data Collection: We use Schneider and Schultes' [5] test code as a starting point. It covers 98 language constructs for the Swift to Kotlin transpilation direction, and 94 language constructs for the Kotlin to Swift transpilation direction. The data for our study was collected by transpiling the input test code using the chosen tools. We transpiled all the Kotlin test code files into Swift and all the Swift test code files into Kotlin. The transpilation using SequalsK [10] was done as follows: for each file in the test code, we copied and pasted the input code into the SequalsK website, and then copied the transpiled code from the output into a separate file. We accessed the CodeLlama-7B-Instruct variant, of size 7 billion parameters, via the Ollama library (version 0.11.6). We created a custom Python script to send a pre-determined prompt to the CodeLlama model containing the input code and save its output into a designated output file. The aim of this study is not to examine how different prompts can be used to produce more optimised results. Therefore, we use the same "vanilla prompt" template as Pan et al. [14] which refers to a prompt that only contains 4 basic pieces of information: instructions in natural language to perform the transpilation task, source language, target language, and the code to be transpiled.

3) Data Analysis: After transpiling the input code, each output was manually compared to its expected output file, and any potential bugs were noted. The expected output files, transpiled by Schneider and Schultes [5], served as ground truth. Following this, the bugs were classified into the bug taxonomy defined by Catolino et al. [15], which provides a general bug taxonomy. Categories were then further refined. Thematic analysis was used for this process [16]. The themes identified were the different root causes for the bugs under the pre-existing umbrella taxonomy. Lastly, the resulting taxonomy was examined to determine which bugs could potentially be fixed using mutation operators. Based on this analysis, mutation operators were proposed to repair the bugs automatically.

4) Large Language Model Set Up: To identify a suitable prompt for CodeLlama, we experimented with zero-shot, one-shot and few-shot prompts, from which a one-shot prompt approach consisting of the task, buggy code and only short descriptions of mutation operators produced better bug fixes. Providing just the buggy code snippet, rather than the full file, also improved results. The final prompt used is listed in Fig. 1. In practice, a fault localisation tool could be integrated in the framework to provide the location of the bug. Finally, all buggy code transpilations were input into the LLM. Outputs were manually analysed to assess if a suitable fix was generated.

III. Results

A. RQ1: Tool Effectiveness

After transpiling the test code, we manually analysed the output for bugs to assess transpilation tool effectiveness. The results are presented in Table I. 'Bugs' are defined as transpiled code that differs from the expected transpilation as indicated in the test code files, which would lead to behaviour that is incorrect in relation to the expected behaviour of the code.

```
This is a Kotlin code snippet that contains one or more bugs. Fix the
bug(s) in the code and return the fixed code snippet.
$SOURCE_CODE

Here are some examples of mutation operators that could be used to fix
the bug(s) in the code:

•       Delete lines of code.
•       Copy lines of code and insert them in other parts of the code.
•       Change the type of a variable.
•       Insert library import statements.
•       Insert skeleton code for Kotlin–specific concepts or keywords,
such as the 'when' expression, 'set' statement, the 'in' or '!in'
keyword and the '?.let' concept.
```

Fig. 1. Prompt Used for Kotlin Code Snippets to Fix Buggy Code

TABLE I
Total Bugs Identified Across All Transpiled Code.

Transpilation Bugs			
	SequalsK	**CodeLlama**	**Total**
Kotlin to Swift	20	27	47
Swift to Kotlin	45	57	102
Total	65	84	149

A total of 149 bugs was identified across all the transpilations. We observed that CodeLlama transpilation ouptputs contained more bugs compared to SequalsK, across both transilation directions. Both tools produced more bugs when translating from Swift to Kotlin than from Kotlin to Swift. Next, we present a few examples of the identified bugs.

Bug: Transpilation of `switch` Statement and `when` Expression In Swift a `switch` statement can be transpiled into a Kotlin `when` or a cascading `if-else` statement. Both tools sometimes handled these cases incorrectly.

Bug: Array/List Types A common bug in both tools and transpilation directions was incorrect transpilation of varying data structure types. In particular, incorrect transpilations for array and list types occurred a total of 14 times.

RQ1: SequalsK outperforms CodeLlama in Kotlin–Swift transpilation tasks, with fewer bugs in both directions (13.42% vs. 18.12% for Kotlin to Swift, and 30.20% vs. 38.26% for Swift to Kotlin). Both tools struggle more with Swift to Kotlin, where bug rates more than double.

B. RQ2: Transpilation Bug Types

The identified bugs all fall under the category of "Program Anomaly" in the general bug taxonomy [15]. Thus, we provide a more detailed taxonomy based on our results.

1) Mobile Transpilation Bug Taxonomy: Fig 2 presents the results of our proposed mobile transpilation bug taxonomy split by transpilation tool and direction. The four main bug categories are: Incorrect Code, Extra Code, Missing Code and Earlier Transpilation Error Carried Forward (bugs caused by earlier bugs in the code). As expected, the 'Incorrect Code' category is the most common reason for a bug, followed by 'Missing Code' and 'Extra Code'. The pattern of frequency is consistent across both transpilation directions and tools.

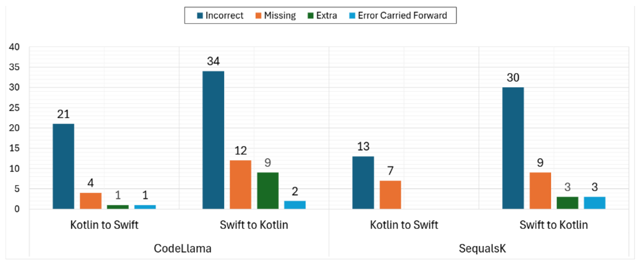

Fig. 2. Bug Distribution By Tool and Transpilation Direction.

Additionally, the SequalsK transpilation tool and the Kotlin to Swift transpilation direction has no bugs caused by Extra Code or Errors Carried Forward and has the least number of bugs overall (only 20), while other configurations have over 30 bugs in one single category.

2) Finer-Grained Sub-Categories of the Bug Taxonomy: Within these categories, there are further finer-grained sub-categories derived that provide more detailed context as detailed in Table II. The most popular sub-category is the ‘Incorrect Type’ at 23 bugs, followed closely by ‘Incorrect Invalid’ (invalid code) at 21 and ‘Incorrect Swift Concepts/Keywords’ at 20. ‘Missing Specific Value’ is the most common sub-category within the ‘Missing Code’ category, whilst ‘Extra General Code’ is the most common for the ‘Extra Code’ category. Unsurprisingly, the sub-categories which are quite specific such as the ‘Extra New Line’, and ‘Extra Cast’ are the least common with at most 2 issues. However, some less-specific sub-categories are also uncommon, such as ‘Extra Function’ which has 1 issue.

RQ2: All bugs fell under the ‘Program Anomaly’ category. They can be further split into: ‘Incorrect Code’ (65.77%), ‘Missing Code’ (21.48%), ‘Extra Code’ (8.72%) and ‘Error Carried Forward’ (4.02%).

C. RQ3: Mutation Operators for Transpilation Bugs

Following on from the taxonomy, we propose mutation operators that could be used to enhance automated repair tooling to fix transpilation bugs between Kotlin and Swift.

Existing generic mutation operators [17] can be used to reduce the bugs occurring during transpilations: *Deletion* can address any bugs caused by ‘extra’ code, making it applicable to 13 bugs. *Insertion* of statements could also be utilised to copy code snippets from the existing program and insert them randomly within the transpiled code. Potentially any bugs caused by ‘missing’ code could be fixed using this mutation operator, making it applicable to 32 bugs. However, these provide a more general rather than strategic approach to reduce the number of bugs.

We identify three novel mutation operators to provide a targeted approach towards reducing the number of transpilation bugs. Firstly, the *Variable Type Mutation* mutation could tackle the common bug seen where the transpiled type of a variable was wrong. The mutation would involve identifying variables in the code, and randomly mutating the type of the variable from a pre-defined list of variable types. Second, the *Library Import Insertion* would involve randomly inserting common import statements into the beginning of the code. The source of the import statements could be a list of common libraries in Swift and Kotlin. Additionally, this could be further customised based on the type of transpilation, and which libraries are more common for certain code tasks. Finally, the *Skeleton Code of Key Concepts/Keywords Insertion* mutation. A common bug was caused by an incorrect transpilation of specific Kotlin and Swift concepts or keywords. For example, the Kotlin `when` expression is usually transpiled into the Swift `switch` statement or a cascading `if-else` statement, which was commonly incorrectly transpiled. This mutation would insert skeleton code of common concepts. Profiling could be used to provide targeted code insertions.

RQ3: Existing generic mutation operators of ‘deletion‘ and ‘insertion’ can be used to reduce the numbers of bugs. Additionally, 3 novel mutation operators have been proposed, that are targeted at specifically reducing transpilation bugs occurring when transpiling between Kotlin and Swift. Together they have potential to fix 101 out of 149 bugs in our dataset.

D. RQ4: Efficacy of Large Language Models

Overall, CodeLlama provided adequate fixes for 12 out of 149 total bugs. Results are shown in Table III. Majority of bug fixes (11 of 12) were for SequalsK transpilations, with only 1 fix for CodeLlama. Fixes introduced no incorrect code, and redundant code was limited to LLM comments. Unfortunately, none of the proposed fixes explicitly made use of the mutation operators included in the prompt. Using larger models and further prompt engineering may improve results. Nonetheless, this study demonstrates that it is possible to integrate an LLM into a framework to reduce the number of bugs in transpilations between Kotlin and Swift.

RQ4: CodeLlama, using our enhanced prompt, provided 12 suitable bug fixes out of 149 tested bugs, with 11 for SequalsK and 1 for CodeLlama transpilations.

IV. Research Agenda

As this study provides an initial exploration of the bugs that occur during transpilation between Kotlin and Swift, it opens several avenues for future research. The taxonomy presented for transpilation bugs helps identify common pain points in transpilation tools and for developers in the mobile industry. Here, we propose mutation operators that target specific bug patterns, which is a first step towards systematically testing and improving transpilers and LLMs in transpilation tasks. These suggested operators can be used to inject mutations, when mutation testing, to assess the quality of code. They are

TABLE II
FINER-GRAINED BUG TAXONOMY SUB-CATEGORIES WITH DESCRIPTION OF EACH CATEGORY AND SUB-CATEGORY.

Category	**Sub-Category**	**Number**	**Description**
Incorrect	Invalid	21	Code that is entirely invalid, with no identifiable atomic issue.
Incorrect	Kotlin Concepts/Keywords	13	Code containing errors specific to Kotlin concepts/keywords.
Incorrect	Object-Oriented	10	Errors in object-oriented concepts (constructors, classes, etc)
Incorrect	Specific Value	2	Very specific value is incorrect.
Incorrect	Swift Concepts/Keywords	20	Code containing errors specific to Swift concepts/keywords.
Incorrect	Type	23	Incorrect code relating to types of variables.
Incorrect	User-Defined Functions	3	Code with errors in transpiling user-defined functions.
Incorrect	Function	6	Incorrect built-in functions used.
Extra	Cast	2	Code with unnecessary type casting.
Extra	Function	1	An extra function wrapper in the code.
Extra	General Code	6	General extra valid code, such as extra methods.
Extra	Invalid	3	Invalid code due to extraneous, non-transpiled additions.
Extra	New Line	1	Extra new line character.
Missing	Imports	8	Missing imports in the transpilation.
Missing	Return	3	Missing code relating to return values
Missing	Object-Oriented	3	Missing object-oriented concepts (constructors, classes, etc)
Missing	Specific Value	9	Missing very specific code, such as a specific value/keyword.
Missing	Transpilation	7	Transpiled code missing corresponding parts from input code.
Missing	Type	2	Missing type specifications.
Error Carried Forward	Earlier Error Carried Forward	6	Issue directly due to an earlier error in transpilation.

TABLE III
RESULTS FOR NUMBER OF BUGS FIXED BY THE LLM

	SequalsK		**CodeLlama**	
	Fixed	Not Fixed	Fixed	Not Fixed
Kotlin to Swift	5	15	0	27
Swift to Kotlin	6	39	1	56
Total	11	54	1	83

equally applicable to be used to detect common bug patterns in transpiled code between Kotlin and Swift. LLMs can fill in the gaps where rule-based tools fall short. Insights from the suggested mutation operators and bug taxonomy can be used to guide both prompt designs and repair workflows in LLMs. This opens a path towards fully-automated transpilation repair in mobile development. Finally, a new framework can be built that automates this workflow, through combining bug taxonomy, mutation-based program repair and LLM-based repair. This framework could serve as a platform for research and tool development to improve transpilation quality.

V. CONCLUSIONS & FUTURE WORK

Transpilation in the mobile industry is an under-explored area of research within software engineering. Here, we provide an initial overview into the bugs occurring when transpiling between Kotlin and Swift using automated tools. We present a taxonomy for the identified bugs and suggestions for novel mutation operators to enhance program repair techniques for fixing transpilation bugs. We identified 149 bugs in our dataset, of which 101 are potentially fixable with the proposed mutation operators. Additionally, we tested the efficacy of the CodeLlama LLM guided by the suggested mutation operators in fixing those bugs, generating 12 suitable fixes. Insights from this work can be used to improve rule-based and LLM-based transpilation approaches and potentially derive a new framework to automate the suggested workflow. This study is a step forward towards fully automated Kotlin-Swift transpilation, and highlights promising directions for future research.

REFERENCES

[1] W. Ahmad et al., "Pricing of mobile phone attributes at the retail level in a developing country: Hedonic analysis," *Telecommunications Policy*, vol. 43, no. 4, pp. 299–309, 2019.

[2] B. Wang et al., "User-Customizable Transpilation of Scripting Languages," *Proc. ACM Program. Lang.*, vol. 7, p. 201229, 2023.

[3] K. H. Ukpabi and A. M. Ibrahim, "Exploring operating system diversity: A comparative analysis of Windows, Mac OS, Android and iOS," *J. of Systematic and Modern Science Research*, 2024.

[4] A. Bastidas Fuertes et al., "Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry," *MDPI Applied Sciences*, vol. 13, no. 6, 2023.

[5] L. Schneider and D. Schultes, "Evaluating Swift-to-Kotlin and Kotlin-to-Swift Transpilers," in *Internat. Conf. on Mobile Soft. Eng. and Systems*. Inst. of Electrical and Electronics Engineers Inc., 2022, pp. 102–106.

[6] D. Ramos et al., "Batfix: Repairing language model-based transpilation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 6, Jun. 2024.

[7] C. Le Goues et al., "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, Nov. 2019.

[8] Q. Zhang et al, "A survey of learning-based automated program repair," *ACM Trans. on Soft. Eng. and Method.*, vol. 33, no. 2, pp. 1–69, 2023.

[9] T. S. Repository, https://github.com/SOLAR-group/mobile-bug-taxonomy-and-repair.

[10] D. Schultes, "SequalsK - A Bidirectional Swift-Kotlin-Transpiler," in *Internat. Conf. on Mobile Soft. Eng. and Systems*, 2021, pp. 73–83.

[11] B. R. et al., "Code Llama: Open foundation models for code," https://arxiv.org/abs/2308.12950, 2024.

[12] Z. Yang et al., "Exploring and Unleashing the Power of Large Language Models in Automated Code Translation," *Proc. ACM Softw. Eng.*, vol. 1, pp. 1585–1608, 2024.

[13] P. Rangeet et al., "Understanding the effectiveness of large language models in code translation," *CoRR*, vol. abs/2308.03109, 2023.

[14] R. Pan et al., "Lost in Translation: A Study of Bugs Introduced by Large Language Models While Translating Code," in *ACM Internat. Conf. on Soft. Eng.*, 2024, pp. 995–1007.

[15] G. Catolino et al., "Not All Bugs Are the Same: Understanding, Characterizing, and Classifying the Root Cause of Bugs," *J. Syst. Softw.*, vol. 152, no. C, p. 165–181, 2019.

[16] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, 2006.

[17] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic Improvement of Software: A Comprehensive Survey," *IEEE TEVC*, vol. 22, pp. 415–432, 2018.