SequalsK—A Bidirectional Swift-Kotlin-Transpiler

Dominik Schultes
Technische Hochschule Mittelhessen
Friedberg, Germany
dominik.schultes@iem.thm.de

This is the preprint version of

D. Schultes, "SequalsK—A Bidirectional Swift-Kotlin-Transpiler," 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), 2021, pp. 73-83, doi: 10.1109/MobileSoft52590.2021.00017.

https://ieeexplore.ieee.org/document/9460946

***Abstract*—Developing two separate versions of an app for iOS and Android causes considerable efforts. Therefore, a lot of cross-platform development frameworks are available that are able to produce apps for both platforms out of a single code base. However, there are tradeoffs that are connected with these frameworks, in particular, a high tool dependency. Therefore, we propose to stick with native development but to take advantage of the shrinking gap between both platforms due to the fact that the current programming languages, Swift for iOS and Kotlin for Android, are considerably more alike than their respective predecessors. Relating to the model-view-controller design pattern, we propose to automatically transpile the *model* part of the app in a *bidirectional* fashion, i.e., from Kotlin to Swift and vice versa. This way, iOS experts can concentrate on optimizing the user interface (view- and controller-part) for iOS, and Android experts can do the same for Android, but all developers can jointly work on the shared model part: each developer can read, correct, and enhance the source code of the model using their own preferred programming language; the resulting version is then transpiled to an equivalent version in the other language.**

We present a working prototype of a transpiler—which we call *SequalsK*—that supports the majority of the important constructs of both languages and is able to generate syntactically and semantically correct Kotlin code out of Swift code *and vice versa*.

In a case study we show that the model part of a board game app can be transpiled in both directions without any limitations. Starting with a working Swift version, the Android version can be derived with little manual effort: the automatically transpiled model part forms 86 percent of the resulting source code.

***Index Terms*—mobile app, transpilers, transcompilers, Swift, Kotlin, cross-platform**

I. Introduction

The mobile operating system market has always been fragmented. For the last couple of years, the situation has been quite clear with only two big players left: in November 2016 the total market share of Android and iOS exceeded 90 % for the first time and has grown to 99.4 % in December 2020 [1]. Thus, from a practical point of view as an app developer, it is nowadays sufficient to concentrate on only two platforms. However, in most cases, neither of these platforms may be neglected. If a developer targeted only iOS, they would lose 72.5 % market share [1]. If only Android was targeted, a significant share of the total purchasing power would be lost since Apple's App Store for iOS generated 87.3 % more in consumer spending than Google's Play Store for Android in 2020 [2].

Both platforms provide software development kits (SDKs) to allow the creation of apps for the respective platform. The targeted hardware and the provided features have many similarities, e.g., that usually every mobile device no matter if running Android or iOS can access GPS data and the respective SDK makes these data available to the app developer. Furthermore, the app development for both platforms has conceptual similarities as well, e.g., that object-oriented programming and design patterns like model-view-controller are applied. Nevertheless, the concrete implementations are quite different. Firstly, different programming languages are used: Java or Kotlin for Android vs. Objective-C or Swift for iOS. Secondly, different APIs are provided, e.g., the actual methods to retrieve GPS data have different names. Thirdly, different tools are integrated into the respective IDE, e.g., for designing the user interface.[1]

This leads to the fact that when developing an app for both platforms, one is usually able to reuse parts of the software architecture, probably a large amount of the assets like images, and some concepts, but from a pure implementation point of view, one has to do the whole work twice. Accordingly, this causes a large total effort. Furthermore, the time to market increases when the platforms are dealt with one after the other—or, if the apps for both platforms are implemented simultaneously, the total effort probably further increases since similar problems might be solved simultaneously by different developers, wasting more time than a successive implementation for both platforms would have cost where one developer could learn from the findings of the other developer. To make things worse, two different code bases typically lead to significant problems w.r.t. maintainability: bug fixing and implementing additional features require that similar steps have to be performed within both implementations. Sooner or later this might lead to inconsistencies between both apps, which is usually not desirable.

In order to avoid all these drawbacks, there is a wide range of cross-platform approaches that allow app development for at least the two major platforms with only one code base.[2] However, often there are tradeoffs, w.r.t. performance and user experience [3], resulting app size [4], degree of maturity and power of the development tools (e.g., the lack of appropriate debugging facilities), robustness of the created apps due to additional sources of errors introduced by the

[1] Conceptual similarities w.r.t. the user interface design, though, have rather grown during recent years, e.g., the *constraint layout* in Android bears a resemblance to the *storyboard* in iOS.

[2] For an overview of the different approaches refer to Section II.

added abstraction layer[3], or—perhaps most importantly—an additional dependency on the used cross-platform framework: if the respective framework does not support a crucial new platform feature or even fully disappears at some point in the future, one might end with a useless code base and might be forced to manually migrate to a different framework.

When looking at these tradeoffs, going one step back and using the platform-specific SDKs appears in a better light, but, of course, we still need to find ways to reduce the high effort of creating and keeping two code bases. The gap w.r.t. the employed programming languages has closed somewhat by introducing Swift in 2014 and making Kotlin the preferred language for Android development in 2019. Even though Objective-C and Java have similarities as well (both are object-oriented and have the programming language C as a common ancestor), the similarities between Swift and Kotlin are considerably bigger: at some points, it is even hard to believe that both languages have been developed independently of each other. Due to the existing similarities, it gets attractive to think of automatically transpiling the code of one language to the other and vice versa. Some language constructs bear such a striking resemblance that even a simple script that applies search-and-replace using regular expressions could be worth giving it a try. But is it really that simple? How promising is the approach of creating a Swift-Kotlin-transpiler? How much development effort could be saved this way?

Our Contributions: We present a new cross-platform development approach that avoids, on the one hand, the tradeoffs of existing cross-platform development frameworks and has, on the other hand, the potential to considerably reduce the usual overhead that arises by creating and maintaining two code bases (Section III). We conduct a comprehensive comparison of language constructs of Swift and Kotlin in order to distinguish between constructs that are identical, very similar, transpilable with some effort, and fundamentally different (Section IV). As the centerpiece of our development approach, we introduce *SequalsK*, the prototype of a *bidirectional* transpiler, i.e., a transpiler that is able to create syntactically and semantically correct Kotlin code out of Swift code *and vice versa* (Section V). In a case study, we apply our approach, and in particular our transpiler, to a board game app in order to analyze the possible savings w.r.t. development effort (Section VI).

II. Related Work

There are many different cross-platform development frameworks that have the common goal to develop apps for multiple platforms from only one code base. We adopt the categorization from [6].[4] The *web approach* summarizes the attempt to offer a web application, developed with standard web technologies (HTML, CSS, JavaScript) and viewed within the normal browser app on the mobile device; usually, modern HTML5 features and some additional frameworks like jQuery Mobile [10] or Sencha Ext JS [11] are used to offer an acceptable user experience on the mobile device. Due to the restricted access rights of the browser, web applications will never be on a par with a mobile app that is installed on the device. The *hybrid approach* uses web technologies as well but bypasses the browser restrictions by packing the web code together with a web view component and a bridge that connects JavaScript function calls with native features in order to form a mobile app that can be regularly installed on the target platforms. Widespread representatives of this approach are, for example, Cordova [12] and Ionic [13]. The *interpreted approach* works by providing runtime engines for the targeted platforms that are able to interpret the same source code written in some language and execute it on the respective platform. Appcelerator [14] and React Native [15], for example, interpret JavaScript code, Flutter [16] interprets Dart code. The *cross-compiled approach* means that code is written in some language and is then processed by different compilers that produce for each target platform a separate, executable unit. For example, Xamarin [17] reads C# source code and outputs, amongst others, apps for Android and iOS; Qt [18] has similar abilities reading C++ source code instead. The *model-driven approach* bears analogy to the cross-compiled approach in the sense that apps for the target platforms are generated from a common source—with the distinction that the common source is not written in a programming language, but in a domain-specific language. MD2 [19], [20] is an example technology for this approach. Disadvantages of all these approaches have already been mentioned in Section I.

A different approach is to develop for one platform natively, i.e., with the suggested programming language using the official SDK, and use a tool to make a compiled version of the code available at the other platform. This way platform-specific parts of the app have to be implemented once more, but platform-independent parts like data structures and business logic can be reused. For example, Kotlin/Native [21] allows the creation of a library out of Kotlin code which can be used by Swift code in an iOS project.[5] By this means, it is possible to get rid of some of the problems of the above mentioned five approaches. At least the code base can be used to directly create an app for one platform so that the tool dependency is somewhat reduced. W.r.t. user experience and performance good results can be expected as well. However, the course of development is not optimal: in the Kotlin/Native example, Swift developers who spot a bug in the provided library cannot fix it themselves but have to ask the Kotlin developers to fix it for them and to provide a new version of the library. They even cannot analyze the source code in the language that they are familiar with. In times of agile software development where "working software [is valued] over comprehensive documentation" [24], it is a drawback if comprehensive, up-to-date documentation is

[3]For example [5] reports inconsistencies that are caused by the cross-platform framework Xamarin.

[4]There are alternative categorizations as well, e.g., [7]–[9].

[5]There are similar considerations and tools for the opposite direction [22], [23].

always compulsory to compensate for the fact that the source code cannot be examined in an appropriate way.

Instead of providing a compiled library, it is an interesting idea to transpile the source code of one programming language to the source code of the programming language of the other platform [25], [26]. This way, all developers can analyze and understand the used code in their own language, i.e., the previously mentioned 'documentation-problem' is avoided. But still, the course of development is not optimal since the developers of one platform depend on the bug fixing and improvement efforts of the developers of the platform that the code originally has been implemented for.

There are several unidirectional transpiler projects either from Kotlin to Swift [27] or from Swift to Kotlin [28], [29]. On the one hand (Swift to Kotlin), in particular *Gryphon*, is quite mature generating code that is—in the best cast—syntactically and semantically correct so that it can be used without manual corrections. On the other hand (Kotlin to Swift), *Kotlift* is less ambitious and it is likely that manual postprocessing is needed, which might be acceptable if the Kotlin code was transpiled only once to Swift. But, in reality, it is much more likely that the Kotlin code is improved and extended from time to time so that the repeated manual postprocessing would get infeasible.

There is work on bidirectional transpilers between Ada and Pascal [30] and between COBOL and C++ [31], which are languages that are quite different from Kotlin and Swift. In both cases, the focus is not on using bidirectional transpilers within a cross-platform software development project.

There are blog articles that compare language constructs of Swift and Kotlin [32], [33], however, they tend to concentrate on the aspects with high similarity and somewhat exclude the difficult parts.

III. Native Cross-Platform Development Approach

Usually, mobile apps—and, generally, applications with a graphical user interface—are developed applying some sort of model-view-controller (MVC) pattern [34], as depicted twice in Fig. 1, once within the top rectangle showing the iOS version of an app, and once within the bottom rectangle showing the analogous Android version.[6] The *model* stands for the state and behavior of the application, in other words, it covers both the structures that hold the application's data and the business logic. The *view* consists of elements that are used to present the application's state to the user and to accept the user's input. The *controller* contains code that reacts to user actions, for example, by calling methods of the model in order to update the application's state. The controller can also be notified by the model that something has changed so that it can update the view accordingly.

[6]Note that there are various variants of the MVC pattern as well as related patterns like model-view-presenter (MVP) [35] or model-view-viewmodel (MVVM) [36]. Here, we adopt the concrete representation and wording from [37]. Also note that with the introduction of SwiftUI [38] the MVC representation of an iOS app might no longer be accurate in the future; however, since we solely concentrate on transpiling the model part, these subtleties are not relevant for this work.

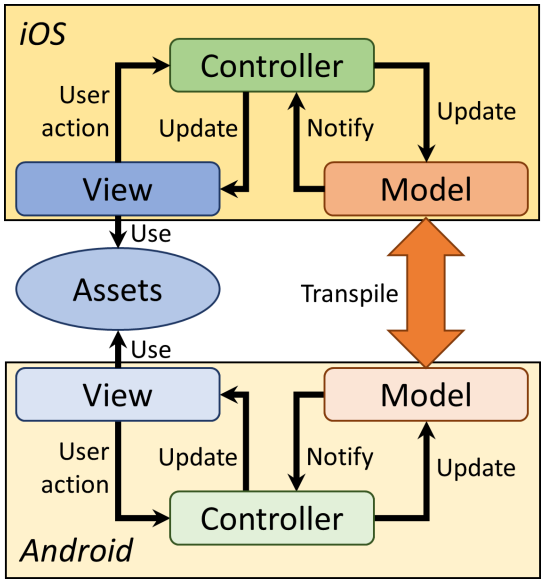

Fig. 1. Overview of the proposed native cross-platform development approach.

Without the double-headed "Transpile" arrow, Fig. 1 represents the traditional way to develop an app for iOS and Android using two separate code bases. In most cases, it is possible to reuse assets, in particular images, for both platforms. Apart from this, there is much work that has to be done twice. Note that while the controllers and views are platform-specific, using specific APIs that are provided by each platform, the models typically only differ by the used programming language.

We propose—in order to avoid the already mentioned disadvantages of cross-platform development frameworks that work with a single code base—to stick with the separate native development relying on two code bases but—in order to reduce the total effort—to introduce a *bidirectional transpiler* that is capable of translating the iOS model written in Swift to the Android model written in Kotlin and vice versa. By this, the effort of writing the model twice can be avoided. Fig. 1 represents this idea by adding the double-headed "Transpile" arrow. A naive counting of the components shows that traditionally seven components (twice model, view, and controller and once the assets) have to be taken care of, while our transpiler approach reduces this to six components since the model has to be developed only once. This might appear as just a small saving, but depending on the extent of the model in relation to the rest of the code this can lead to significant savings, cf. Section VI.

Provided that we have a bidirectional transpiler that produces syntactically and semantically correct code, we can organize the development of a two-platform app as follows. We work with one team of iOS/Swift experts and one team of Android/Kotlin experts. Each team is able to make the most of the respective platform using modern features and working with the official tools. The work on the model part of the app can be shared in an arbitrary way among both teams. Each

team can enhance the model, add additional features, and fix bugs in their own preferred programming language. Each team can use the model directly from their respective controller code, e.g., call methods of the model, and, if necessary, examine the model code during debugging, regardless of whether the relevant part was originally developed in their own language or in the other one. Both the Swift and the Kotlin translation of the model should be kept under version control. When a change is committed, the transpiler is used to translate the new version to the other language; the outcome is committed as well. This way, it makes no difference whether a Swift programmer changes Swift code and then another Swift programmer works with the updated code or whether a Kotlin programmer changes Kotlin code and then a Swift programmer works with the transpiled code. Both teams have equal rights, they can work in parallel so that the time to market for none of the platforms is delayed. No team must wait for the other team to implement an important feature to the model since each team can improve the model on its own.

IV. Swift vs. Kotlin

A complete and systematic comparison of the programming languages Swift and Kotlin would exceed the scope of this paper. Both are complex languages offering a lot of features whose specifications would have to be examined in detail. Fortunately, in every-day programming usually only a subset of all features that a programming language provides is used so that it can be reasonable to concentrate on a frequently-used subset first. To avoid a random choice, in this section we deal with all aspects that are covered in the introductory chapter "A Swift Tour" of the official Swift documentation [39], assuming that the most important and most interesting features have been selected for that chapter. In addition, in Section V-B we mention some additional features that are supported by our transpiler.

In general, we concentrate on aspects that belong to the programming language itself, i.e., that can be found in the grammar of the language. We do not want to compare the APIs of both environments. However, there is no strict boundary between language and API, e.g., in Swift arrays and dictionaries are part of the grammar of the language, while in Kotlin these features are part of the API. Hence, at some points, we have to touch the API as well.

The following subsections match the sections of "A Swift Tour" [39]. The findings are summed up in Tab. I. We distinguish between constructs

- that are identical in both languages ($=$), e.g., `var x=1`,
- that are very similar ($\approx$) in the sense that a simple string replacement (without using regular expressions) would be sufficient to transpile the code, e.g., `let x=1` vs. `val x=1`,
- that can be transpiled with some effort ($\leftrightarrow$), and
- that are fundamentally different ($\neq$).

A. Simple Values

In contrast to Kotlin, in Swift we do not need an explicit main program but can write the code of the main program just directly in the global scope ($\leftrightarrow$). Simple output to the console is very similar in both languages ($\approx$). Declaring and initializing a variable as well as reassigning a new value is identical ($=$). W.r.t. constants both languages use different keywords ($\approx$). However, the difference at this point gets considerably bigger when dealing with arrays ($\leftrightarrow$): in Swift an array is a value type so for a constant array it is not allowed to change one of its elements, in Kotlin an array is a reference type and only reassigning another array is forbidden for a constant array while changing the contents is allowed. Basic types (`Int`, `Double`, `Boolean`), type inference, and providing explicit types are very similar concepts in both languages ($\approx$). However, w.r.t. implicit casts there are differences, e.g., in Swift an `Int` is automatically cast to `Double` if required, in Kotlin an explicit cast is necessary ($\leftrightarrow$). Converting an integer to a string works differently ($\leftrightarrow$). Both languages support string interpolation with small differences in syntax ($\leftrightarrow$). Multiline strings are supported in both languages—with some subtle differences, for example, w.r.t. the indentation rules ($\leftrightarrow$).

In Swift, arrays and dictionaries are built into the language with a concise syntax; in Kotlin only the indexing suffix in order to access an element of a collection is part of the grammar; apart from that, arrays (with fixed size), lists (with variable size), and dictionaries (rather called maps in Kotlin) are represented by classes of the standard API ($\leftrightarrow$). Reading or changing an element of a collection works in the same way in Swift and Kotlin ($=$).

B. Control Flow

Most common operators are identical ($=$) or similar (e.g., the nil-coalescing/elvis operator) ($\approx$), however, Swift does not support increment and decrement operators ($\leftrightarrow$).

The if-statement can be transpiled with little effort ($\leftrightarrow$). Optional binding, i.e., checking whether an optional variable is not nil and introducing a new variable that holds the unwrapped contents, is done within an if-statement in Swift, while Kotlin uses a quite different way ($\leftrightarrow$). Conceptually, the switch/when-statements of Swift and Kotlin are very similar, but the syntax has some differences ($\leftrightarrow$); the quite special where-clause of Swift has no correspondent in Kotlin ($\neq$).

Using a for-loop in order to iterate through an array or through a range of integers in forward direction can be easily transpiled; the same applies to iterating through all key-value-pairs of a dictionary, although the technical background is quite different: Kotlin employs a so-called destructuring declaration at this point, while Swift employs a tuple, a built-in feature of the language, which is of more general use ($\leftrightarrow$).

While-loops and repeat/do-while-loops have only small syntactical differences ($\leftrightarrow$).

TABLE I
COMPARISON OF SWIFT AND KOTLIN LANGUAGE CONSTRUCTS (SECTION IV) AND LIST OF FEATURES FULLY (✓✓) OR PARTLY (✓) SUPPORTED BY THE SEQUALSK TRANSPILER (SECTION V).

construct	similarity	supported
Simple Values		
form the main program	↔	✓
print something to the console	≈	✓✓
declare a variable	=	✓✓
declare a constant	≈	✓✓
mutate an array	↔	✓
handle basic types	≈	✓
implicitly cast an `Int` to a `Double`	↔	–
convert an `Int` to a `String`	↔	✓
interpolate a string	↔	✓
handle multiline strings	↔	✓
create an array, a list, and a dictionary	↔	✓
access an element of a collection	=	✓✓
Control Flow		
most common operators	=	✓✓
nil-coalescing/elvis operator	≈	✓✓
increment/decrement operators	↔	✓✓
if-statement	↔	✓✓
optional binding	↔	✓✓
switch/when-statement	↔	✓✓
where-clause within switch	≠	–
for-loop	↔	✓✓
while-loop and repeat/do-while-loop	↔	✓✓
Functions and Closures		
declare a function	≈	✓✓
return from a function	=	✓✓
call a function	↔	✓✓
argument labels vs. parameter names	↔	✓✓
overload a function by different argument labels	≠	–
tuple as return value	↔	✓
nested functions	=	✓✓
functions as first-class citizens	↔	✓✓
closures with $\leq$ one default parameter name	↔	✓✓
closures with $>$ one default parameter name	↔	–
Objects and Classes		
declare and instantiate a class	=	✓✓
access properties and functions of an object	=	✓✓

construct	similarity	supported
Objects and Classes (continued)		
initializers / constructors	↔	✓
class inheritance	↔	✓✓
override a function	↔	✓✓
computed properties	↔	✓✓
property observers	↔	✓
optional chaining / safe calls	=	✓✓
Enumerations and Structures		
enumerations (incl. properties and functions)	↔	✓✓
raw values of enumeration cases	↔	✓
switch/when on enumeration cases	↔	✓✓
init an enumeration by raw value	↔	✓✓
associated values of enumeration instances	↔	–
read-only usage of structs / data classes	↔	✓✓
mutating usage of structs / data classes	↔	–
Protocols and Extensions		
declare a protocol / interface	↔	✓✓
implement a protocol by a class	↔	✓✓
implement a protocol by a struct	↔	–
use the name of a protocol as a type	=	✓✓
extensions (except for subsequent special cases)	↔	✓
use extension to implement a protocol	≠	–
use extension to modify an `Int` value	≠	–
Error Handling		
declare a throwable construct	↔	✓✓
group related errors	↔	–
mark a function that throws an error	↔	✓✓
throw an error	=	✓✓
catch an error	↔	✓
place cleanup code	↔	–
handle a function result as optional (`try?`)	↔	✓✓
Generics		
declare a generic function	↔	✓✓
declare a generic class or enumeration	=	✓✓
enumerations with associated values of generic types	↔	–
declare a generic protocol / interface	↔	✓✓
specify generic constraints	↔	–

C. Functions and Closures

Declaring a function with or without parameters and with or without a return value is very similar in both languages (≈). The return-statement is even identical (=). Calling a function, however, exhibits an important difference: In Swift, each parameter value usually must be preceded by the corresponding parameter name (↔). To make matters more difficult, in Swift one can distinguish between an argument label, which is used when calling a function, and a different parameter name, which is used within the function body; Kotlin does not support such a distinction (↔). In Swift, a function can be overloaded by another function with the same name, but different argument labels; in Kotlin this is not possible since Kotlin does not share the concept of argument labels (≠).

In Swift, multiple values can be returned from a function using a tuple—a quite general concept already mentioned in Section IV-B, which is not directly supported in Kotlin; it can be emulated by introducing a corresponding data class (↔).

The concept of nested functions is identical in both languages (=).

Both in Swift and in Kotlin, functions are first-class citizens and can be used as a type for parameters, return values, and variables, in other words, functions can be passed to functions, returned from functions, and stored in variables; the syntactical differences are rather small (↔).

With closures (Swift) and lambda expressions (Kotlin), both languages have similar concepts of a special kind of anonymous functions, i.e., blocks of code that can be passed to higher-order functions and called later; the syntax of both languages has some similarities as well as some differences at this point (↔). Kotlin supports only one default parameter name for a lambda expression (`it`), while Swift allows using `$0`, `$1`, `$2`, ... as default names for any number of parameters of a closure (↔).

D. Objects and Classes

Declaring a class and instantiating it, i.e., creating an object, is identical in both languages (=). Accessing a property or calling a function of an object works identically as well (=). Swift and Kotlin provide different options in order to initialize an object. At first glance, the concepts—designated and convenience initializers on the one hand (Swift) and primary and secondary constructors on the other hand (Kotlin)—might seem quite similar, however, there are several more or less

subtle differences that have to be considered when finding the correct counterpart for an initialization construct in the other language (↔).

With respect to inheritance, there are both similarities, e.g., writing the name of the superclass after a colon, and differences, e.g., using the keyword `open` in the superclass declaration in Kotlin (↔). The same observation applies to overriding functions (↔).

Both languages support computed properties, i.e., properties that do not just represent a stored value, but that allow arbitrary getter and setter code that defines what happens when the property is read or written to; the syntax, however, is not identical (↔). In addition, Swift allows so-called property observers (`willSet` and `didSet`); Kotlin does not provide a direct counterpart, but writing an appropriate setter can emulate property observers in most cases (↔).

Optional chaining (Swift) and safe calls (Kotlin), i.e., using `?.` to safely access a function or property of a variable that might be nil, work identically in both languages (=).

E. Enumerations and Structures

Both languages support enumerations and allow to add properties and functions; the basic syntax is somewhat different (↔). In Swift, for each enumeration case a raw value can be defined; this can be emulated in Kotlin as well (↔); Swift handles raw values in a more convenient way (e.g., automatic numbering), while Kotlin offers more flexibility (e.g., more than one raw value). A switch/when-statement can be applied to enumeration cases in both languages (↔). A special failable initializer can be used in Swift to try to retrieve an enumeration case that matches a given raw value; in Kotlin this can be emulated by providing an appropriate static invoke operator (↔).

In Swift, each particular instance of an enumeration case may be enriched by associated values; Kotlin does not support associated values for enumerations; however, the behavior can be emulated by introducing for each enumeration case an own class with the respective associated-value declaration nested in one sealed class that represents the enumeration type (↔).

In addition to normal classes and enumerations, both languages include one additional class-like construct: structs (Swift) and data classes (Kotlin). While the typical use cases of these constructs are often similar, the technical details are different: structs are passed by value, data classes are passed by reference. If these constructs are used in a read-only fashion, they are virtually equivalent (↔); otherwise, transpiling might be possible (e.g., by introducing—if required—explicit deep copying in Kotlin), but very difficult (↔).

F. Protocols and Extensions

A protocol in Swift and an interface in Kotlin have identical semantics; the way to distinguish between constant and variable properties differs (↔). Implementing a protocol/interface is similar, but in Kotlin implemented properties and functions have to be preceded by the `override` keyword (↔). Structs and data classes can also implement protocols/interfaces (↔). The name of a protocol/interface can be used as a type in both languages (=).

Extensions can be used in both languages to add further functionality to existing types (↔). In Swift, an extension can be used to make an existing type implement a protocol; this is not supported in Kotlin (≠). Furthermore, in contrast to Swift, in Kotlin an extension cannot be used to modify the value of a primitive data type like an `Int` (≠).

G. Error Handling

Error (Swift) and exception (Kotlin) handling are quite similar concepts, but there are several differences when looking at the details. In Swift, constructs that can be thrown and caught must implement the `Error` protocol, while in Kotlin subclasses of `Throwable`—or, in most cases, of `Exception`—are used (↔). In Swift, it is usual to group related errors within an enumeration; in Kotlin this can be emulated by separate classes nested in a sealed class (↔), analogously to dealing with associated values of an enumeration (cf. Section IV-E). Functions that may throw an error must be marked by the `throws` keyword in Swift, while this is not necessary in Kotlin (↔).[7] Throwing an error or an exception is done by using the `throw` keyword (=).

The `do-catch`- (Swift) and `try-catch`-blocks (Kotlin) work analogously; however, in Swift, the call of a function that may throw an error must be explicitly marked by a preceding `try` keyword (↔). In Kotlin, a `try`-block can be extended by a `finally`-part that contains cleanup code; in Swift cleanup code can be placed anywhere within a discrete `defer`-block (↔). In Swift, `try?` in front of a function call can be used to conveniently ignore an error and get `nil` instead; since `try` can be used in Kotlin as expression, this behavior can be emulated (↔).

H. Generics

Both languages support generic functions, using angle brackets to enclose generic parameters; only the position within the function declaration differs (↔). Both Swift and Kotlin support generic types, in particular, generic classes and enumerations, which can be declared with exactly the same syntax (=). Note that the already mentioned fact that Kotlin does not directly support associated values (cf. Section IV-E) makes it difficult to emulate an enumeration with an associated value of a generic type (↔).

Dealing with generic protocols or generic interfaces, respectively, differs significantly: while Kotlin, when declaring a generic interface, uses the same syntax as for generic class declarations, Swift features a quite special syntax at this point by using an `associatedtype`-declaration within the body of the protocol (↔). This difference makes a transpilation difficult in many respects, in particular when specifying generic constraints (↔).

[7] In contrast to Java, Kotlin only supports *unchecked exceptions* so that the `throws` keyword becomes obsolete.

I. Summary

Plain summing up yields 11 identical, 5 similar, 45 transpilable-with-more-effort-than-string-replacement, and 4 fundamentally different constructs. On the one hand, this shows that Swift and Kotlin actually are different languages despite the fact that you can find some examples where the languages seem to be almost equal. Thus, just working with "find and replace" will not lead to satisfying results.[8] On the other hand, there is rarely an important construct in one language that has no suitable correspondent in the other language. We can conclude that creating a bidirectional transpiler that covers all important constructs seems possible, but non-trivial.

V. The SequalsK Transpiler

A. Architecture and Implementation

We strive for developing one bidirectional transpiler and not two separate unidirectional transpilers because we want to reuse some components and we want to create a single tool that is able to transpile the source code of one language to the other language and back again in order to provide instant feedback on whether the current code can be processed in both directions successfully. Note that this goal is considerably more challenging than developing a unidirectional transpiler since we cannot just rely on the existing compiler of one platform because closely building on the compiler of one language would make processing the other language much more difficult. Instead, we develop the bidirectional transpiler basically from scratch, except for the lexer and parser part, where ANTLR [40] is used as valuable help. Fig. 2 gives an overview of our SequalsK transpiler, the used technologies, and the data flow. The starting point for the development are the grammars of both languages, each separated in a lexer and a parser part (depicted in yellow in Fig. 2). In the case of Kotlin, the official grammar was directly available in ANTLR4 notation [41]. In the case of Swift, an unofficial version relating to Swift 3.0.1 was available [42], which needed some corrections and to be split up into a lexer and a parser part. Using ANTLR we generated lexer and parser code in Java (depicted in orange). The central class `Transpiler` holds references to the lexers and parsers and coordinates the whole processing of the given source codes. The main part of the transpiler is written in Kotlin (depicted in various shades of green, one shade for common classes, a darker one for classes that process Swift input, and a lighter one for classes that process Kotlin input).

The source code that should be transpiled is, firstly, processed by the corresponding lexer, which retrieves the tokens; the tokens are passed on to the parser, which, secondly, builds the parse tree (the processed data is depicted in blue in Fig. 2, a dark shade for Swift, a light shade for Kotlin). Our main contribution are the Swift and Kotlin parser visitors, which, thirdly, come into action at each relevant node of the parse tree, while the tree is traversed. Each parser visitor uses its own language-specific type-inference mechanism in order to infer the types of declared variables. This is non-trivial since both languages while being statically typed do not require giving explicit type annotations within the declarations. The functionality of a symbol table is needed by both visitors so that this component can be reused. For example, when a visitor processes a node of the parse tree that represents a variable declaration, it infers the type of the variable and stores the name and the type in its own instance of the symbol-table data structure. Later on, the entries of the symbol table can be retrieved, e.g., in case of function overloading, in order to decide which function is actually called depending on the types of the parameters—this is important to pick the correct argument labels when transpiling from Kotlin to Swift.

The parser visitors also use common helper classes that sup-

[8]Note that the differences would be considerably bigger if Objective-C was compared to Java.

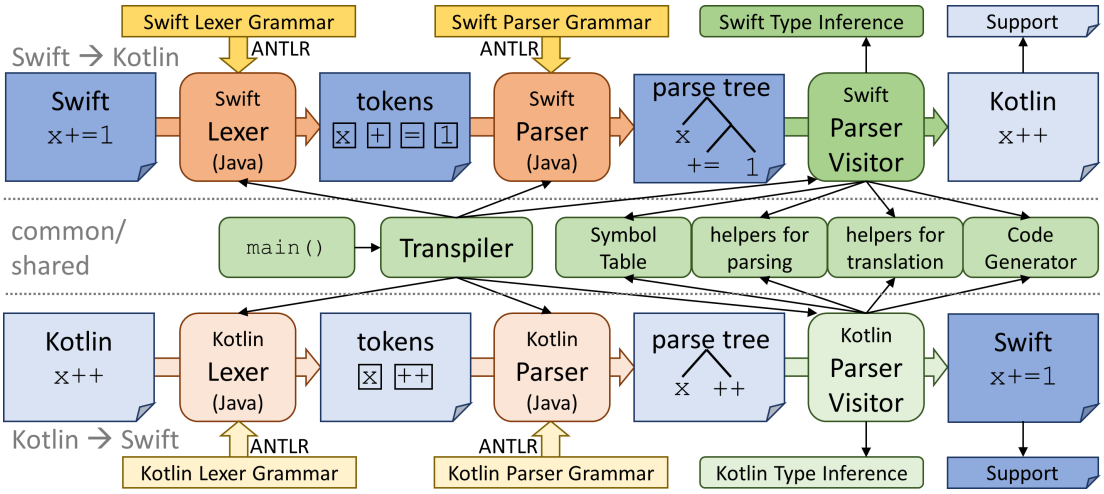

Fig. 2. Overview of the SequalsK transpiler. The upper third shows the needed components and the data flow of transpiling Swift source code to Kotlin. The lower third deals with the opposite direction. The middle third contains the components that are shared by both parts of the bidirectional transpiler. A broad arrow means "generates", a normal thin arrow means "uses".

port the parsing process, in particular, by navigating the parse tree to look up certain elements, and helper classes that support the translation process, for example, a simple map that helps to replace `Bool` (Swift) by `Boolean` (Kotlin) and vice versa. Furthermore, a separate class, the code generator, is used to write the transpiled code, also dealing with white-space and indentation issues. The resulting, transpiled code might not work on its own. For some aspects we need some supporting code in the target language. Therefore, the transpiled code should be viewed together with a static support file. For example, the Swift-support-in-Kotlin file contains the line `annotation class argLabel(val name: String)`. When Swift code is transpiled to Kotlin the original Swift argument labels are kept in annotations within the Kotlin code. This keeps the Kotlin code readable and usable, but also allows a translation back to Swift and restoring the original argument labels.

B. Supported Features

Currently, the SequalsK transpiler is in the state of a working prototype. Tab. I was already presented in Section IV to summarize the comparison of the considered language constructs. Now, we revisit the table and add a statement (column "supported"),

- whether our transpiler supports the construct without any limitations (✓✓),
- with some limitations that are probably negligible in most normal use cases (✓), or
- whether our transpiler does not support the construct (or only rudimentary) (–). Note that in case of a construct marked by ↔, this means that this feature can be added in a future release of the transpiler, while in case of a construct marked by ≠, this means that this feature would probably never be added, which implies that this particular language construct must be avoided in a cross-platform development project.[9]

In addition to the constructs shown in Tab. I, which covers all aspects mentioned in "A Swift Tour" [39], the SequalsK transpiler also supports the following features: static properties and functions (Swift) / companion objects (Kotlin), subscripts (Swift) / `get` and `set` operators (Kotlin), and `inout` parameters (Swift).

In the following, we give some selected examples of source code, exactly as it is produced by the SequalsK transpiler.[10] Fig. 3 demonstrates how a primary constructor is transpiled. Note that it would be possible to make the Kotlin code look more like the Swift code by separately defining a constant property `x` and by using the keyword `constructor` to define a constructor. However, this would be less elegant and we strive for code that most programmers of the target language will prefer. Otherwise, a Kotlin programmer might manually improve the code by introducing a Kotlin-style primary constructor and would get annoyed if a future transpilation from Swift destroyed their work.

[9]Fortunately, all ≠-constructs can be considered as quite exotic and, probably, can be easily avoided.

[10]We just made a few adjustments to indentation and line breaks to save some space.

```
// Swift
class C {
    let x: Int
    init(x: Int) {
        self.x = x
    }
}
```

```
// Kotlin
class C(val x: Int) {
}
```

Fig. 3. Example: a primary constructor.

```
// Swift
func greet(_ person: String, from hometown: String) {
    print("Hello \(person)!")
    print("Glad you could visit from \(hometown).")
}

func main() {
    greet("Bill", from: "Cupertino")
}
main()

// Kotlin
fun greet(@argLabel("_") person: String, @argLabel("from")
     hometown: String) {
    println("Hello ${person}!")
    println("Glad you could visit from ${hometown}.")
}

fun main() {
    greet("Bill", "Cupertino")
}
```

Fig. 4. Example: argument labels.

As mentioned at the end of Section V-A, the generated output has to be treated in combination with a small support file. The already mentioned example of dealing with argument labels is depicted in Fig. 4. This example emphasizes the fact that bidirectional transpilation is particularly challenging. If we transpiled only from Swift to Kotlin, we just could forget about the argument labels. But, since we want to allow Kotlin developers to enhance the code as well, we need the shown mechanism to restore the original argument labels so that the Swift developers do not lose their accustomed code style.

Swift's property observers `willSet` and `didSet` can be rewritten in Kotlin as two sections of a setter as depicted in Fig. 5. The transpiler introduces a new auxiliary variable `⍉newValue` whose name starts with a rarely used Unicode character to avoid conflicts with existing variables and to help the transpiler to restore the original Swift code out of the produced Kotlin code. Note that a Kotlin programmer can add arbitrary code above or below the line `field = ⍉newValue` and the code is automatically assigned to the correct property observer when transpiling back to Swift.

The example shown in Fig. 6 demonstrates that although the syntax of a generic protocol (Swift) is quite different from the syntax of a generic interface (Kotlin), it is possible to

```swift
// Swift
class C {
  var x: Int = 0 {
    willSet {
      print("old: \(x)")
    }

    didSet {
      print("new: \(x)")
    }
  }
}
func main() {
  C().x = 42
}
main()
```

```kotlin
// Kotlin
class C {
  var x: Int = 0
    set(@newValue) {
      println("old: ${field}")

      field = @newValue

      println("new: ${field}")
    }
}

fun main() {
  C().x = 42
}
```

Fig. 5. Example: property observers.

```swift
// Swift
protocol I {
  associatedtype T
  func f() -> T
}

class C: I {
  typealias T = Int
  func f() -> T {
    return 42
  }
}
```

```kotlin
// Kotlin
interface I<T> {

  fun f() : T
}

class C : I<Int> {

  override fun f() : Int {
    return 42
  }
}
```

Fig. 6. Example: a generic protocol/interface.

transpile the code in both directions, offering developers at both sides straightforward and maintainable code. Note that some subtle differences like the fact that in Kotlin a method that implements an interface needs the `override` keyword makes the development of a transpiler more challenging.

VI. Case Study: Board Game

In the context of a lecture on Swift, a board game app had been developed, covering the games Checkers and Reversi; the app consisted of 13 source code files, summing up to 854 lines of code. The app offers a single-player mode, ensures that only valid moves can be performed, and contains a straightforward implementation of the minimax algorithm [43, p. 165] in order to provide an AI as the opponent. The code is structured according to the model-view-controller design pattern (cf. Section III); the model covers data structures, in particular, to store the state of the board, the game logic, in particular, the functionality to determine the set of allowed moves and to check which player has won, and the implementation of the AI. The app works without any limitations. Fig. 7 represents the model part of the app as an UML class diagram. The controller, which is not depicted in the diagram, only interacts with the model via a reference of type `Game`; depending on the chosen game, the controller creates an appropriate instance

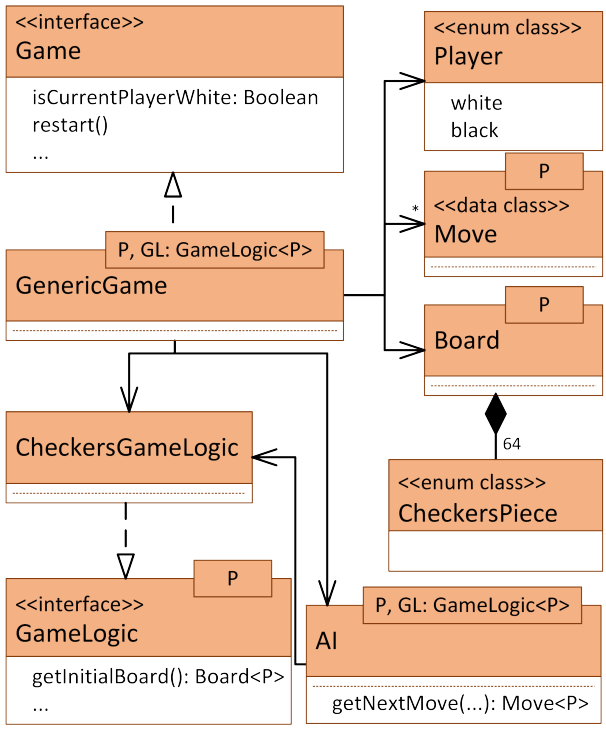

Fig. 7. Simplified UML class diagram of the model of the board game app.

of `GenericGame`.[11]

Since the app originally had been intended as an example to demonstrate a wide range of Swift features, it seems to be a suitable test case for a transpiler. Thus, the first version of the SequalsK transpiler has been developed targeting the board game example. The original Swift code was only slightly simplified resulting in 833 lines of code; 77 % of the source code belongs to the model part.

The SequalsK transpiler is able to transpile the model part of the Swift code to syntactically and semantically correct Kotlin code, achieving good code quality (w.r.t. readability and maintainability) as well. After transpiling the code of the model to Kotlin, the author spent only one hour and 43 minutes in order to manually add Kotlin code for the view and controller parts, resulting in a fully working Android app with the same functionality as the original iOS app. The resulting Android app consists of 762 lines of Kotlin code; 86 % of the source code belongs to the model part and, thus, has been automatically generated by transpiling from Swift. In other words, 86 % of the effort of writing an Android version of the board game app could be saved in terms of lines of code. At

[11] In the diagram we have arbitrarily chosen to use Kotlin keywords like `interface` and so on. Its purpose is to give a rough overview of the architecture of the board game app, in particular, to demonstrate that an interesting mixture of language constructs (generic classes, enum classes, a data class, interfaces, ...) is used. However, the details of the app are not in focus. Therefore, several simplifications have been made: only very few methods are included, only the classes needed for one concrete game, namely Checkers, are shown, and we draw direct associations to `CheckersPiece` and `CheckersGameLogic`, just assuming that `CheckersPiece` is substituted for `P` in generic classes and `CheckersGameLogic` for `GL`.

that point, transpiling back from Kotlin to Swift was supported as well for that version of the board game.

In the second stage, in the Android app Chess was added as a third game, written in Kotlin. Note that the game logic of Chess is somewhat more complex than that of Checkers and Reversi. In total, the resulting app consists of 1 221 lines of code. The bidirectional transpiler did not work out of the box, but some features had to be added to cover aspects introduced by the new Chess part of the app. To be precise: 11 % of the lines of code of the resulting transpiler have been added or changed during the second stage. Now, it is possible to transpile the model part of the app including Chess in both directions, without spending any manual rectification work.

Summing up, we find that the SequalsK transpiler can be used to fully transpile the model part of a medium complex app and, thus, to save a high percentage of the effort of porting an app from one platform to another.

VII. Conclusion

The previous section demonstrated that the proposed native cross-platform development approach looks promising. If we are willing to avoid rather exotic programming constructs, the existing prototype of the SequalsK transpiler is already able to successfully transpile the model part of an app from Swift to Kotlin and vice versa. Summing up, to implement the proposed approach in a real cross-platform development project, we need the following ingredients:

- a team that is composed of two subteams—iOS/Swift experts on the one hand, and Android/Kotlin experts on the other hand—(cf. Section III),
- project management that coordinates the platform-independent parts of the development process, in particular the work on the model part,
- the bidirectional SequalsK transpiler (cf. Section V),
- a project infrastructure, in particular, a version control system, that allows to easily update the model in both languages (cf. Section III),
- support files for both platforms that are needed to make some transpiler output compilable (cf. Section V), and
- particular coding conventions that help to avoid constructs that are difficult to transpile (cf. Tab. I). With further enhancements of the transpiler, some limitations noted in the coding conventions may be removed in the future.[12]

The proposed approach avoids disadvantages of existing cross-platform development frameworks. Most importantly, we get rid of the tool dependency: if we developed a huge cross-platform project using the SequalsK transpiler and building two large code bases over the years and if the SequalsK transpiler was not maintained and was not be able to work with future versions of Swift or Kotlin, then we still would be able to use and maintain our *native* code bases in the future and we would only lose the advantage of maintaining the model only once. In contrast, if we developed a huge code base with, for example, the programming language Dart in combination with the framework Flutter and Flutter reached its end of life, we would practically have to throw the whole code base away.[13]

The observed savings rate of 86 % in the board game case study may be considered as the best case. In most apps, the fraction of platform-specific code will be higher and, thus, the savings will be smaller. However, savings can be improved, if we provide a common interface (with different underlying platform-specific implementations) for frequently used features like dealing with HTTP requests or retrieving location updates [25]. In a sense, this idea has similarities to existing cross-platform frameworks that provide a common interface (for example, in JavaScript) for accessing various native features. However, we must be aware of the fact that providing such an abstraction layer will require more frequent updates in the future since the underlying platform-specific APIs usually change more rapidly over time than the pure programming languages. Thus, there will be a considerable dependency of an app development project on a SequalsK abstraction layer. Still, we will have the great advantage that an Android developer will be able to immediately integrate a new Android feature—and the same applies to an iOS developer—even if it is not directly supported by the abstraction layer as everyone works with native technology, i.e., with Kotlin and Swift, so that there will not be any technological gap.

When looking at the savings, one should consider not only the hard facts but also psychological aspects: implementing platform-specific parts twice striving for an optimal user experience on each platform is much more satisfying then manually implementing the model part twice being aware of the fact that essentially the same things are programmed—almost mindlessly—twice in two programming languages that look very similar.

Future Work: Besides the obvious next steps—adding further features to the transpiler and performing more and larger case studies—we have already started a bunch of concrete subprojects, which we consider interesting: Firstly, we want to provide a plugin for Android Studio, in particular, to integrate immediate feedback what language constructs can be transpiled without causing problems.[14] Secondly, it would be nice to automatically transpile unit tests from one platform to the other as well. This way, we could obtain additional assurance that the transpiler produces semantically correct code. Thirdly, we aim at providing a common interface to frequently used concurrency operations, like doing computational work in several threads and collecting the results in the main thread.

Finally, we want to transpile the transpiler itself in order to integrate the functionality in Xcode as well.

[12] Such coding conventions should not be viewed as mere limitations caused solely by missing features of the transpiler. Some rules may also help to improve the code quality. For example, using only `Double` literals and no `Int` literals in the context of `Double` variables or parameters does not only simplify the transpilation (since the Kotlin compiler will not accept `Int` literals at this point), but also leads to (slightly) more readable code because it is clearly visible that the further processing is done with a floating-point number and not with an integer.

[13] Note that this argument is not new but is prominently mentioned in [26] as well, however, only in the context of a unidirectional transpiler, which has significant drawbacks as described in Section II.

[14] [29] contains a similar plugin for Xcode.

References

[1] StatCounter, "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide/2016 (accessed 01/2021).

[2] SensorTower, "Global consumer spending in mobile apps reached a record $111 billion in 2020, up 30% from 2019," https://sensortower.com/blog/app-revenue-and-downloads-2020 (accessed 01/2021), 2021.

[3] P. Que, X. Guo, and M. Zhu, "A comprehensive comparison between hybrid and native app paradigms," in *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2016, pp. 611–614.

[4] A. Ebone, Y. Tan, and X. Jia, "A performance evaluation of cross-platform mobile application development approaches," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, pp. 92–93.

[5] N. Boushehrinejadmoradi, V. Ganapathy, S. Nagarakatte, and L. Iftode, "Testing cross-platform mobile app development frameworks," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 441–451.

[6] M. Latif, Y. Lakhrissi, E. H. Nfaoui, and N. Es-Sbai, "Cross platform approach for mobile application development: A survey," in *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, 2016, pp. 1–5.

[7] R. Nunkesser, "Beyond web/native/hybrid: A new taxonomy for mobile app development," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, pp. 214–218.

[8] K. Shah, H. Sinha, and P. Mishra, "Analysis of cross-platform mobile app development tools," in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, 2019, pp. 1–7.

[9] S. Charkaoui, Z. Adraoui, and E. H. Benlahmar, "Cross-platform mobile development approaches," in *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, 2014, pp. 188–191.

[10] "jQuery Mobile," https://jquerymobile.com/ (accessed 01/2021).

[11] "Sencha Ext JS," https://www.sencha.com/products/extjs/ (accessed 01/2021).

[12] "Apache Cordova," https://cordova.apache.org/ (accessed 01/2021).

[13] "Ionic," https://ionicframework.com/ (accessed 01/2021).

[14] "Appcelerator," https://www.appcelerator.com/ (accessed 01/2021).

[15] "React Native," https://reactnative.dev/ (accessed 01/2021).

[16] "Flutter," https://flutter.dev/ (accessed 01/2021).

[17] "Xamarin," https://dotnet.microsoft.com/apps/xamarin (accessed 01/2021).

[18] "Qt," https://www.qt.io/ (accessed 01/2021).

[19] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "Cross-platform model-driven development of mobile applications with md2," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 526533. [Online]. Available: https://doi.org/10.1145/2480362.2480464

[20] "MD2," https://wwu-pi.github.io/md2-framework/ (accessed 01/2021).

[21] "Kotlin/Native," https://kotlinlang.org/docs/reference/native-overview.html (accessed 01/2021).

[22] E. Wing, "Swift on Android: The future of cross-platform programming?" https://academy.realm.io/posts/swift-on-android/ (accessed 01/2021), 2017.

[23] "SCADE," https://www.scade.io/ (accessed 01/2021).

[24] K. B. et al., "Manifesto for agile software development," https://agilemanifesto.org/ (accessed 01/2021), 2001.

[25] A. G. Olloqui, "Playtomic's shared architecture using Swift and Kotlin," https://dev.to/playtomic/playtomics-shared-architecture-using-swift-and-kotlin-320b (accessed 01/2021), 2018.

[26] V. J. Vendramini, A. Goldman, and G. Mounié, "Improving mobile app development using transpilers with maintainable outputs," ser. SBES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 354363. [Online]. Available: https://doi.org/10.1145/3422392.3422426

[27] "Kotlift," https://github.com/studo-app/Kotlift/ (accessed 01/2021), 2020.

[28] "SwiftKotlin," https://github.com/angelolloqui/SwiftKotlin (accessed 01/2021), 2020.

[29] "Gryphon," https://vinivendra.github.io/Gryphon/ (accessed 01/2021).

[30] P. F. Albrecht, P. E. Garrison, S. L. Graham, R. H. Hyerle, P. Ip, and B. K. Brückner, "Source-to-source translation: Ada to Pascal and Pascal to Ada," in *Proceedings of the ACM-SIGPLAN Symposium on The ADA Programming Language*, ser. SIGPLAN '80. New York, NY, USA: Association for Computing Machinery, 1980, p. 183193. [Online]. Available: https://doi.org/10.1145/800004.807949

[31] S. Blomberg and J. Severin, "Creating a bi-directional source-to-source compiler using MDE transformation techniques," Master's thesis, Chalmers University of Technology, 2017.

[32] A. Hussain, "A comparison of Swift and Kotlin languages," https://www.raywenderlich.com/6754-a-comparison-of-swift-and-kotlin-languages (accessed 01/2021), 2018.

[33] "Swift is like Kotlin," http://nilhcem.com/swift-is-like-kotlin/ (accessed 01/2021).

[34] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *J. Object Oriented Program.*, vol. 1, no. 3, p. 2649, Aug. 1988.

[35] M. Potel, "MVP: Model-View-Presenter; the Taligent programming model for C++ and Java," http://www.wildcrest.com/Potel/Portfolio/mvp.pdf (accessed 01/2021), 1996.

[36] J. Smith, "Patterns – WPF apps with the Model-View-ViewModel design pattern," *MSDN Magazine*, vol. 24, no. 02, Feb. 2009. [Online]. Available: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

[37] "Model-View-Controller," https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html (accessed 01/2021), 2018.

[38] "SwiftUI," https://developer.apple.com/xcode/swiftui/ (accessed 01/2021).

[39] Apple Inc., "The Swift programming language (Swift 5.3)," https://docs.swift.org/swift-book/ (accessed 12/2020), 2020.

[40] T. Parr, *The Definitive ANTLR 4 Reference*. O'Reilly UK Ltd., 2013.

[41] "Kotlin grammar," https://kotlinlang.org/docs/reference/grammar.html (accessed 10/2019).

[42] "Swift grammar," https://github.com/antlr/grammars-v4/tree/master/swift/swift3 (accessed 10/2019).

[43] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Education Limited, 2016.