Cross-language Clone Detection for Mobile Apps

Stephannie Jimenez[1], Gordana Rakic[2], Silvia Takahashi[1], Nicolás Cardozo[1]

[1]Systems and Computing Engineering Department - Universidad de los Andes, Colombia

[2]Faculty of Sciences - University of Novi Sad, Serbia

{s.jimenez16,stakahas,n.cardozo}@uniandes.edu.co, gordana.rakic@dmi.uns.ac.rs

Abstract. *Clone detection provides insight about replicated fragments in a code base. With the rise of multi-language code bases, new techniques addressing cross-language code clone detection enable the analysis of polyglot systems. Such techniques have not yet been applied to the mobile apps' domain, which are naturally polyglot. Native mobile app developers must synchronize their code base in at least two different programming languages. App synchronization is a difficult and time-consuming maintenance task, as features can rapidly diverge between platforms, and feature identification must be performed manually. Our goal is to provide an analysis framework to reduce the impact of app synchronization. A first step in this direction consists on a structural algorithm for cross-language clone detection exploiting the idea behind enriched concrete syntax trees. Such trees are used as a common intermediate representation built from programming languages' grammars, to detect similarities between app code bases. Our technique finds code similarities with* 79% *precision for controlled tests where Type 1-3 clones are manually injected for the analysis of both single- and cross-language cases for Kotlin and Dart. We evaluate our tool on a corpus of 52 mobile apps identifying code similarities with a precision of* 65% *to* 84% *for the full application logic.*

1. Introduction

In software systems it is usual to find repeated code fragments, also known as code clones. Large modern software systems' code base has 20% to 50% clones [Ain et al. 2019]. Normally, clones are innocuous, but add complexity to a code base and induce higher maintenance costs. For instance, the time to add a feature or fix a bug increases as the code requiring changes may be replicated across multiple modules, requiring the propagation of the changes to all clone locations. Moreover, assuring that the applied changes are consistent across clones remains a challenge [Mondal et al. 2020].

Clones refer to similar but not necessarily identical code fragments. Clones are classified into four hierarchical types: *Type 1* or *exact clones*, refer to identical code fragments not including code comments, layout, or whitespaces. *Type 2* or *renamed clones*, refer to code fragments that have a similar structure or equivalent syntax without including identifiers, types, or literals. *Type 3* or *near miss clones*, refer to fragments that have been altered further, such as adding or removing statements and changes in whitespaces, identifiers, layout, comments, and types. *Type 4* or *semantic clones*, refer to code fragments that are functionally similar, but implemented as syntactic variants [Ain et al. 2019]. Figure 1 exemplifies the differences among clone types in a multi-language setting for Kotlin[1] and

[1]https://kotlinlang.org

Dart.[2] Line 1 shows a Type 1 clone for an assignment of variable `i`. Line 3 shows a Type 2 clone for an assignment where the variable names differ. Line 5 shows a Type 3 clone as the increment operation changes its form. Lines 7-9 show a Type 4 clone where the syntax changes completely but both code fragments increase the value of a variable by 9.

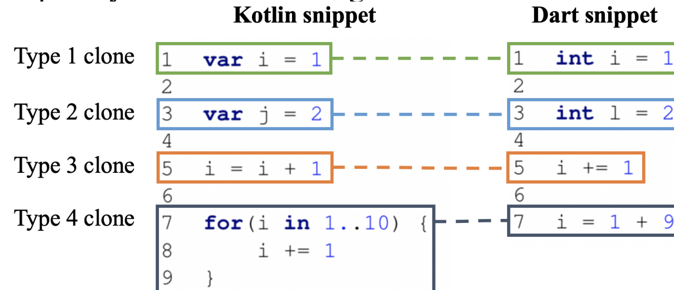

Figure 1. Clone type examples between Kotlin and Dart code bases.

In addition to the four clone types, clones may also be grouped in two families, *structural clones* and *function clones*. Structural clones refer to code fragments that follow the syntactic structure of a given language with a syntactic boundary. Function clones are limited to the procedure or function granularity level [Ain et al. 2019].

Clone detection techniques can be advantageous in code plagiarism, intellectual property protection, or code quality analysis [Ain et al. 2019]. Another example of the usefulness of clone detection tools is that of bug fixing. As mentioned previously, fixing a bug on a cloned functionality is costly. Identifying all similar code snippets to a bug will reduce the technical debt for the development team, as they can incorporate the fix in all existing clones, or refactor the code to reuse a single instance of the clone.

In mobile app development, it is common to find separate code bases using different programming languages to target mobile platforms natively (*i.e.,* iOS and Android), while keeping the functionality between platforms as close as possible. However, app features across platforms often differ due to the separate development teams targeting each platform, and the available platform APIs. We highlight three challenges with maintaining separate code bases for different platforms. First, developers need to ensure that the features across the platforms are equivalent. Second, bug fixes knowledge needs to be transferred across platforms. Third, the maintenance costs of two development teams for each platform is high. To address these challenges, cross-platform code generation frameworks are becoming increasingly popular. However, such approaches may result in a limited use of specific native APIs that may not translate easily across platforms [Patkar et al. 2020].

As a counterpart to multi-platform code generation, the use of a tool to help development teams synchronize feature evolution across mobile platforms is desired. Furthermore, we argue that a cross-language analysis tool that keeps track of equivalent code fragments constitutes a first step to achieve this goal. However, currently no such tool exists. This paper presents the first algorithm and tool for cross-language clone detection for mobile apps, called OUT OF STEP. The main contributions of OUT OF STEP are:

- A generalized model for Enriched Concrete Syntax Trees (eCSTs) (Section 2.1).
- A new clone detection algorithm and metric based on structural clone detection using eCSTs to abstract commonalities across languages (Sections 2.2-2.4).
- A corpus for the evaluation of cross-language clone detection including basic examples, 30 sorting algorithms exemplars, and 52 medium and large mobile apps (Section 3).

[2] `https://dart.dev`

2. OUT OF STEP

This section describes our proposed approach for cross-language clone detection. OUT OF STEP follows a structured approach for clone detection based on the comparison of eCSTs generated from the grammars of modern mobile programming languages as Kotlin and Dart.[3] First, we present the generalized definition of eCSTs. Second, we describe the underlying algorithm and metric to compare eCST structures used to determine clones.

2.1. Enriched Concrete Syntax Trees

To compare of code bases written in different languages we first build an intermediate representation based on the languages' grammars. For our representation, we extend eCSTs by introducing new node types, which allow us to capture the general abstractions of the languages used for mobile app development. Concretely, we use the ANTLR4 [Parr 2013] parse generator to generate the eCSTs focusing on the standard grammar definitions of Kotlin, and Dart. This is applicable to other languages, as Swift, Java, or C++.

2.1.1. eCST Definition

Given that the generated syntax trees may result in highly divergent tree structures, we define a tree structure that is able to capture more information of the input source code. Our solution extends the original proposal of eCST [Rakić and Budimac 2013], with *universal*, *advance*, and *stop* nodes, to uniformly shape trees, structure grammar elements as a group, and ease sub-tree structure comparisons.

Originally, eCSTs are extracted from the generated Abstract Syntax Tree (AST) by the parser. eCSTs introduce universal nodes to shape trees for a Concrete Syntax Tree (CST) or Parse Tree (PT) [Budimac et al. 2012, Vislavski et al. 2018]. In our case, eCSTs are generated directly from PTs and contain a subset of the original universal nodes, as well as a set of mobile app specific nodes (introduced as domain specific universal nodes [Rakić and Budimac 2013]).

Every syntactic rule defined in the languages' grammar is translated to one node. Each node type of the eCST is designed to be as generic as possible, with the objective that rules from different languages generate the same universal node types. Yet, we are required to faithfully represent all expressions defined in a program.

The eCST structure extends the original AST structure by holding more information at nodes, in pursuit of standardizing programming languages to a critical set of features (*e.g.,* grammar concepts). In order to achieve the required behavior, we extend the standard ANTLR4 parsers and lexers for each of the languages. Using this approach, we are no longer required to build programs' ASTs to then translate them to the eCST. Rather, we build the eCSTs directly from the PT generated for each language.

Additionally, eCST nodes keep a keyword token stored as plain text, and a mapping to the line and column of the original source code file in which the keyword appears. With this information, we are able to map the final clones to the original source code file from the tree representation. Moreover, having the actual keyword text helps us recognize

[3] available at: `https://github.com/FLAGlab/CloneDetection/`

the appropriate category of the clones (*i.e.,* Type 1, 2 or 3). As an example, if a node is of type `comparison_operator`, the keyword text information in the node token should be one of the <, <=, >, or >= operators.

Figure 2 shows the tree structure including the new node information. Node types are shown to their right, and the token is shown inside each node. Tokens' line and column are omitted from the figure for clarity. Universal nodes ease tree comparison and analysis by making their structures more similar. Flat structures, as Dart's, now contain information on eCST nodes that will help us map clones (*e.g.,* `assignment_operator`).

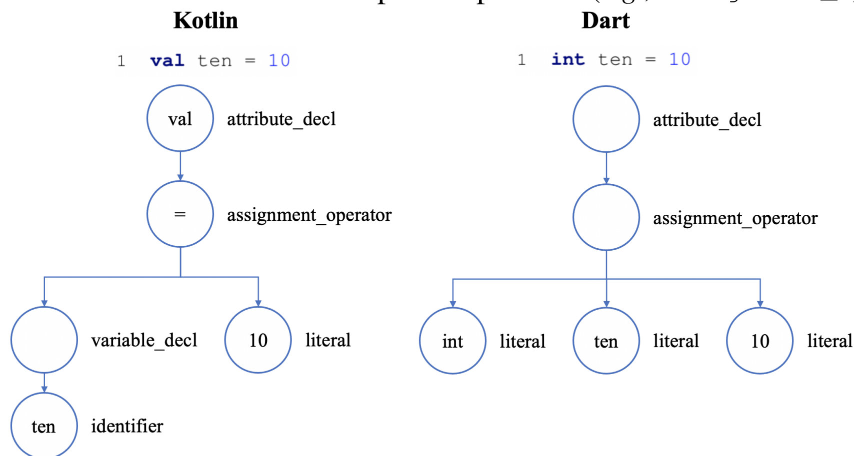

Figure 2. eCST examples for a Kotlin (left) and Dart (right)

In our extension of the eCST, we introduce *advance nodes*. The purpose of these nodes is to group structures of the grammar that should be together. For example, to group the body of the **`for`** `loop`, increasing the trees's hierarchy, as in Figure 3. Advance node `body`, in the loop Dart example, helps us create a parent structure, making the grammar more hierarchical than it originally is. The grammar elements represented with advance nodes include `type`, `body`, and `expression`. Advance nodes are used as information sources, but they are not taken into account in the actual clone detection. A full list of the nodes introduced for eCSTs is available in the project's documentation.

Given that clone types exclude comments, layout, and white spaces, these elements are excluded in the generation of eCSTs. The parser focuses on grammar rules to generate the eCST from the associated code, as in Figure 3.

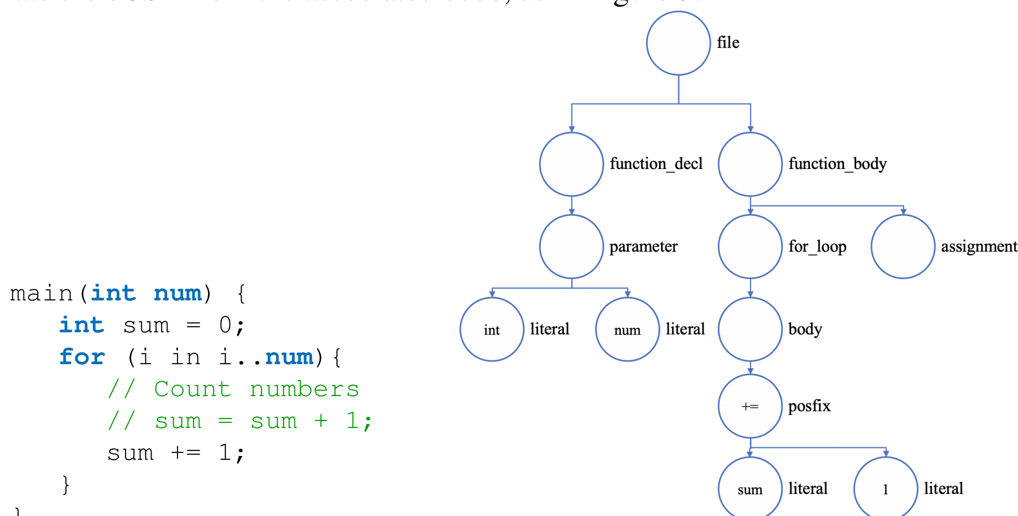

Figure 3. Increased hierarchy using advance nodes for the Dart code for a `loop`

2.1.2. eCST Comparison

Multiple tree comparison algorithms exist in the literature (*cf.* Section 4), the approach taken in OUT OF STEP is to find similar subtree structures between eCST pairs. Under this approach, a pair of similar trees may not be completely identical in their node types, but the types may be closely related. This resemblance is defined via a new mapping, where one type of node can map to multiple other nodes. For example, the similar nodes for the type `parameter_list` will include `value_parameter_list` and itself. All relationships between nodes are established in pursuit of finding Type 3 clones. The complete list of nodes' similarity mapping is also available in the repository's documentation.

Additionally, we define *stop nodes*. These nodes are of special importance in the tree comparison process, as they break down the complete eCST into smaller subtrees to ease comparison. The universal node types that are represented as *stop nodes* are: *function declaration*,*function body*, *class declaration*, and *class body*. As these structures are normally present in scripts, the detector can use them to identify clones, independent and unbiased of their position in the eCST's hierarchy.

The motivation of stop nodes' fragmentation is centered in the idea that clones can be disperse throughout the files. This means that is not necessary for the first function of the first file to map to a clone in the first function on the second file. In this way, having separated substructures of the eCST enables us to compare them among each other and find clones that could map to different locations across source code files.

2.2. Code Clones Detection

The generality of the eCST intermediate representation enables us to map code written in any programming language to this structure. That is, any two eCST trees use the same set of universal nodes making them comparable and agnostic to the specific programming language they abstract.

Using the fragmentation technique introduced with *stop nodes*, multiple subtrees become comparable. For every pair of trees their nodes are visited to see if they are clones of each other, based on the similarity mapping. Once two nodes' types are deemed equivalent, we compare their tokens for (simple) equality. In the case where the type of the nodes is equivalent and the token is exactly the same, we mark them as Type 1 clones. If the type of the nodes is equivalent but the tokens are not equal, we mark them as Type 2 clones. If the nodes are only similar, then they are Type 3 clones.

2.3. Filter Final Code Clones

As we are able to differentiate among clone types, we posit a similarity metric for code fragments based on the number of clones detected. This metric is the weighted sum of the total number of clones found for each type, where their importance follows the restriction that clone types are weighted as Type $1 >$ Type $2 >$ Type 3, as shown in Equation (1).

$$m = 3 * T_1 + 2 * T_2 + T_3 \tag{1}$$

This metric is defined for filtering the possible clones detected in the comparison of all the possible pair-wise combinations of the subtrees. In this way, we are able to define the closest pair among all the possibilities. Our metric shows the most identical clones, and gives precedence to exact clones over others clone types.

For complex and hierarchical structures, a merging mechanism is used where we define if the root node of a tree is similar to some other node by counting the number of clones that are within its children nodes. In this way, we are able to recognize not only the clones that are in the tree leafs, but also clones in intermediate nodes.

Figure 4 exemplifies how clone type weight broadcasting helps us determine when a subtree is more similar to another. In this example, the leaf nodes broadcast their type weights up to the `func_decl` nodes. This way, the corresponding function clone pairs are based on leaf nodes' precedence by their broadcasting rather than by their parent nodes.

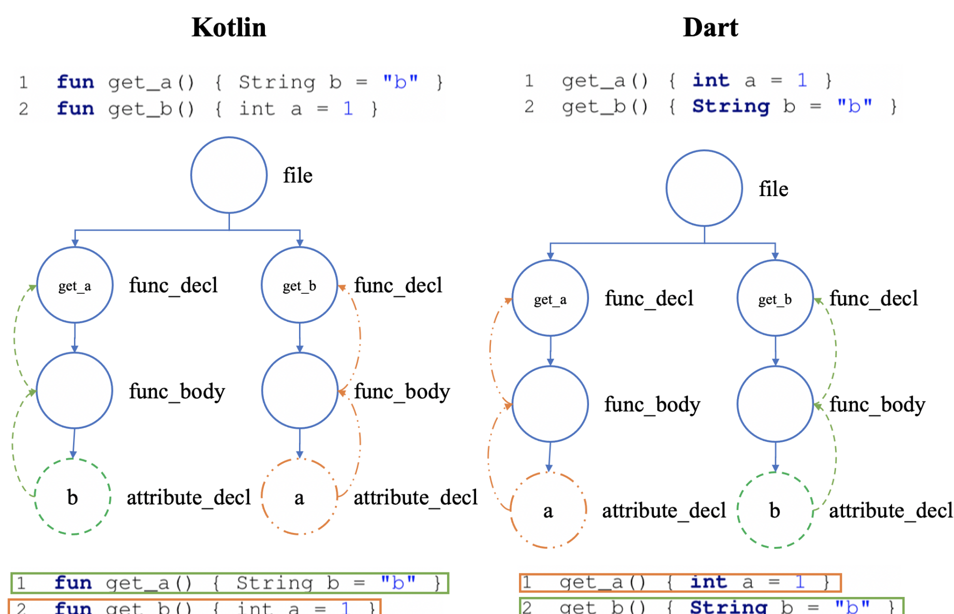

Figure 4. Broadcast method determining the similarity of sub-tree structures

The algorithm returns a list of all detected clones. Each clone is defined as a pair of tokens with a mapping to the original files, the actual string, line and column number, eCST node type, and clone type. All analysis information is reported back to users.

2.4. Out of Step Implementation

The implementation of our proposed approach (*i.e.,* tool and algorithm) exhibits different improvements when compared to other clone detection tools. In this section we describe the details of Out of Step in comparison with other tools.

A key feature of Out of Step, not present in any other tool, is the support for complete programming projects. Normally, existing tools receive as input two specific source files to compare, which might be limiting for large code bases, where users may want to detect clones across full programming projects (*e.g.,* to check feature compatibility, or plagiarism verification). Out of Step can take multiple files as input, as well as whole projects. This enhancement is presented considering the exploration of mobile apps, where developers may need to compare the complete application source code. To attain this need, given a complete project as input, Out of Step performs an exhaustive pair-wise comparison of all the possible source code files, finding all possible clones in the application, which comes with a high complexity ($O(n^3)$) of the comparison algorithm.

Additionally, the Out of Step analysis parametrizes the amount of clones to analyze in the algorithm –that is, to view all detected clones, or to use clone filtering.

This can be useful when the motivation of using OUT OF STEP is to detect repeated code fragments rather than just the most identical fragments in the code base.

Finally, we isolate grammars, with their parsers and lexers, so that adding a new programming languages is completely modular, requiring new folder with the grammar definition to include it in the tool The grammar definition must follow the ANTLR4 specification, as it is the dependency we use to automatically generate langauges' lexer and parser, otherwise a custom lexer and parser must be provided. To complete the integration of new languages, we require a manual mapping between the language's grammar rules and the universal nodes. Not all the grammar rules need to generate a node in the eCST; this depends on the desired granularity of generated trees.

3. OUT OF STEP Evaluation

We evaluate OUT OF STEP in two phases. First we validate clone detection correctness by means of the basic language features for Kotlin, and Dart. Second, we validate the effectiveness of OUT OF STEP to analyze complete mobile app projects. We close this section with a discussion about our results.[4]

3.1. Clone Detection in Structural Elements

To evaluate our clone detection algorithm we first focus in analyzing the basic abstractions across Kotlin and Dart. The goal of this evaluation is to correctly identify clones using general universal nodes in the base case examples of the languages. Due to space constraints, here we show only the case for loops and classes in Kotlin, and Dart. We introduce Type 1 and Type 2 clones to our examples, to verify their correct detection.

3.1.1. Loop nodes

Managing loops is an interesting case for universal nodes. Loop abstractions are defined using different keywords in Kotlin, Dart, and Swift (*e.g.,* `for`, `while`). In OUT OF STEP we process loop abstractions as semantically equivalent, and define them as Type 3 clones. To evaluate this, we manually inject Type 1 and Type 3 clones in our tests.

Snippet 1. Kotlin for loop - version A

```
1 var sum = 0;
2 for (i in 1..100)
3   sum = sum + i
```

Snippet 2. Kotlin while loop - version B

```
1 var sum = 0
2 var i = 1
3 while (i <= 100) {
4   sum = sum + i
5   i = i + 1
6 }
```

Snippet 3. Dart for loop - version A

```
1 int sum = 0;
2 for (int i = 1; i <= 100; i++)
3   sum = sum + i;
```

Snippet 4. Dart while loop - version B

```
1 int sum = 0;
2 int i = 1;
3 while ( i <= 100) {
4   sum = sum + i;
5   i = i + 1;
6 }
```

[4] The full evaluation is available as an online appendix `https://flaglab.github.io/CloneDetection/`. All evaluation data is available at: `https://doi.org/10.5281/zenodo.5228822`

The analysis for Kotlin (Snippets 1 and 2), and Dart (Snippets 3 and 4) in the same language results in two Type 3 clones. These match the `for` and `while` statements and the `i` variable declaration. The clone for the postfix statement `i++` and the variable assignment `i = i + 1` is not correctly detected as a Type 3 clone due to their node representations in the eCST, an `assignment_operator` unary node, and a binary node respectively. Nonetheless, we identify the complete `for` and `while` blocks as clones of each other, as the algorithm broadcasts clones inside the body of blocks. Additionally, OUT OF STEP identifies the declarations and assignment of variables as Type 1 clones in Line 1 of both snippets, and Line 3 of version A with Line 4 of version B.

The cross-language comparison between code versions A and B is also successful in detecting the corresponding clones. This is because the node information and the eCST structure are the same for the two code snippets. Every node is matched to its corresponding node on the other eCST, even though the programming languages differ. When comparing any A vs any B versions, we find that there are Type 3 clones for the `for` and `while` statements, and the definition of variable `i`. Additionally, we can find the Type 1 clones for the declaration and assignment of the `sum` variable, as before.

In the loop analysis, OUT OF STEP finds a couple of Type 2 false positives, due to the two assignments that are present in versions B of the code. In this case, the detection algorithm points to the body of the `for` and `while` statements to be clones. This happens because the intermediate type node for both of them is the same. The exact same behavior takes place with the assignment before the `loop` statement in both cases.

3.2. Clone Detection in Classes and Functions

Given that our proposed algorithm detects Type 1, 2, and 3 clones in simple code snippets, we turn our attention to evaluate its performance with more complex code structures that involve nested eCSTs with a large variety of universal nodes.

3.2.1. Classes

To identify clones in classes and objects, we implement a program that creates and instantiates a class in the main function. Type 1 clones are inserted in this example for versions A (Snippets 5 and 7), and B (Snippets 6 and 8) for the two languages.

Snippet 5. Kotlin class definition - version A

```
1 fun main(args: Array<String>){
2   Person A = Person("A")
3   Person B = Person("B")
4 }
5 class Person(val name : String){
6   fun present() = "Yes"
7   fun greet(other: String) = other
8 }
```

Snippet 6. Kotlin class definition - version B

```
1  class Person(val name: String){
2    fun present(){
3      return "Yes"
4    }
5    fun greet(other: String){
6      return other
7    }
8  }
9  fun main(args: Array<String>){
10   Person A = Person("A")
11   Person B = Person("B")
12 }
```

Snippet 7. Dart class definition - version A

```
1 main(){
2   Person A = Person("A");
3   Person B = Person("B");
4 }
5 class Person{
6   Person(String name){
7     this.name = name;
8   }
9   String present(){
10    return "Yes";
11  }
12  void greet(String other){
13    return other;
14  }
15 }
```

Snippet 8. Dart class definition - version B

```
class NestingPerson { class Person{
    Person(String name){
      this.name = name;
    }
    String present(){
        return "Yes";
    }
    void greet(String other){
        return other;
    }
} }
main(){
    Person A = Person("A");
    Person B = Person("B");
}
```

The version of the class definitions are similar to their counterparts in the other language. We use two different syntax forms for Kotlin to show that we support the latest major version of the language. Additionally, we swap the order of the class and main declarations in the file and scope the `Person` class differently in Dart. This, to test our algorithm is agnostic to the elements' definition order and scope. OUT OF STEP is able to find exact clones in different file locations, scopes, or eCST hierachy.

The analysis of the two versions for a single language yields a complete match of Type 1 clones throughout the document. This is true even for Kotlin, where we use different syntactic elements to define the same class. When comparing Kotlin versions A and B, we also identify a clone in the definition of the function `greet` and `main`. This is because of the similarity in their definition and structure making them a false positive.

In the cross language analysis, when versions B are compared OUT OF STEP performs better than with versions A. This is due to the compressed syntax of Kotlin, which changes slightly the structure of the eCST. Notwithstanding, in general, our algorithm can detect the exact same classes and the `main` function.

This experiment reinforces the case where the code is exactly the same but it differs in definition order and scope. This means our algorithm detects clones using the eCST structure even if the root nodes of the clones are at different tree levels.

3.3. Clone Detection in Mobile Apps

We further evaluate the clone detection for complete mobile apps. To do this, we built a corpus of 52 functionally equivalent apps implemented in Kotlin (26) and Dart (26), spanning different application domains. The apps are extracted from two sources: 40 apps come from of a mobile app development course taught at Unandes between 2018 and 2021. For the course each app must be developed for Android and iOS, normally using two development teams of 2-3 students per language/platform. The remaining 12 apps are extracted from GitHub. We make sure these repositories are maintained by the same development team or are under the same organization, to assure feature equivalence.

We verify that all apps compile and run using the standard IDEs for each language, *i.e.,* Android Studio for Kotlin, and the Flutter framework for Dart. The selected apps for

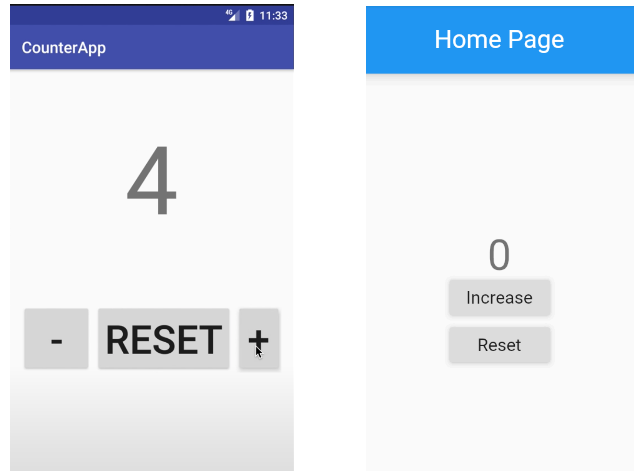

Figure 5. Counter application screenshots

our evaluation vary in application domain (covering 9 domains), and size of the code bases. Since the previous evaluation scenarios used small codebases, we are interested in medium to large apps for this evaluation (with an average of 3100 LOC per application) spanning over several files (an average of 41 files per app). We only take into account the main logic of the project, not including any presentation, configuration, or GUI files.

The analysis between mobile apps is exemplified by a counter application developed by the authors (bold in Table 1). The app counts the number of times a button on the screen is pressed. A screenshot for the Kotlin and Dart versions is shown in Figure 5.

The differences between the apps are immediate. First, the Kotlin version has one extra feature than its Dart counterpart, as users can reduce the counter. However, we expect for the clone detector to be able to find similar features that exist between the two code bases. This follows from the algorithm's clone detection through a pairwise comparison of all files and the functions therein. The result of such exhaustive comparison results in the detection of a large amount of clones.

Snippet 9. Counter app main function in Kotlin

```
class MainActivity:AppCompatActivity(){
   var count = 0
   override fun onCreate(savedInstanceState:Bundle?){
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     textView.text = count.toString()
   }
}
```

Snippet 10. Counter app main function in Dart

```
class MyApp extends StatelessWidget{
  @override Widget build(BuildContext context){
   return MaterialApp(title: 'Flutter Demo',
     theme: ThemeData(primarySwatch: Colors.blue),
     home: MyHomePage(title: 'Home Page'),
   );}
}
```

Our algorithm is capable to find one correct Type 3 clone across the whole code

base (that in Snippets 9 and 10). However the algorithms finds common nodes across the two implementations resulting in many small clones identified for variables and single lines of code. This causes over 60 false positives upon the manual inspection of the code. The high rate of false positives is due, in part, to the frameworks used to implement each of the mobile apps. The implementation frameworks have their own independent APIs, which change the functions used to achieve a particular task in the different programming languages. This has an important impact in the way in which we build the eCST. For example, Snippet 11 shows the signature for the functions used to retrieve an app's splash screen. These functions are semantically equivalent. However, their syntactic difference implies a difference in the the eCST's location and shape. A similar analysis from the manual inspection of OUT OF STEP's results on the other evaluated apps confirms the behavior of our tool, obtaining a precision of 0.65 to 0.84 for mobile app clone detection.

Snippet 11. Kotlin's and Dart's API usage

```
public final SplashScreen getSplashScreen() //Kotlin code
public SplashScreen provideSplashScreen() //Dart code
```

Table 1 shows the clones found by OUT OF STEP in the evaluation of mobile apps. Given the size of the apps, we do not disaggregate Type 2 and Type 3 clones or identify false positives, as these tasks are manual.

Table 1. Cross-language clone detection results for the mobile apps corpus

App Type	Avg. LOC		# of Files		Total	Type 1	Type 2 & 3
	Dart	Kotlin	Dart	Kotlin			
Shopping	1220	3180	6	51	192395	66566	125829
Health	5463	4776	46	48	733716	167673	566043
Health	2956	2236	29	21	110573	27768	82805
Games	2499	3408	20	40	260244	65166	195121
Productivity	4220	3321	85	47	584825	161981	422844
Shopping	3449	1913	74	29	240201	57909	182386
Library	3909	2432	30	32	219373	51978	167395
Shopping	3468	2951	39	84	327299	90354	236945
Shopping	8139	3771	72	84	803354	210749	592605
Health	3660	1836	28	40	106931	27068	79863
Health	4215	1087	39	23	136150	34199	101951
Bookings	4953	3433	44	50	463125	112528	350597
Services	3927	2700	27	31	178192	38384	139808
Lifestyle	3123	1547	21	26	92305	27354	64951
Productivity	2429	1652	21	26	140574	31716	108858
Bookings	1045	1982	25	67	83825	26741	57084
Discounts	2380	1781	37	27	120956	30090	90866
Pets	5754	2615	41	51	250769	70000	180769
Pets	3039	1724	23	26	96517	24828	71689
Bookings	5490	2538	58	81	564947	150226	415011
Productivity	136	104	1	1	149	23	126
Productivity	144	104	1	1	149	23	126
Productivity	**57**	**23**	**1**	**1**	**90**	**25**	**65**
Information	367	2383	6	47	60215	25139	35076
Lifestyle	510	1831	14	59	55591	18471	37120
SmartHome	759	3248	9	33	111810	28052	83758

3.4. Discussion

Running OUT OF STEP on different types of projects across languages for mobile platforms shows the effective clone identification even when dealing with complex project structures and large code bases. Section 3.1 verifies that the basic structures of the two languages are correctly identified as clones. Moreover, from the results in Section 3.2, our algorithm obtains a precision between $[0.65, 0.84]$ for large code bases, demonstrating the correct identification of code clones across languages uniformly in the two data sets.

In the case of mobile apps, we observe that OUT OF STEP finds clones covering complete code bases, as direct consequence of the exhaustive behavior of our algorithm. However, we identify two limitations in the clone detection for mobile apps. First, eCST nodes sometimes lack information to be able to determine whether a code structure can be marked as Type 2 or Type 3 clone. This can reduce the precision of our algorithm, as well as increase required manual verifications. Second, for mobile apps, OUT OF STEP works strictly with the standard definition of the programming languages. As a consequence we do not take into account additional UI or configuration files (*e.g.,* `.xml` files) that developers may use to define the apps' layout. Therefore, we do not detect clones of GUI elements because they are not programmatically defined in the code base. GUI clone identification can give way to further feature divergence analysis across applications.

To enhance the clone detection for mobile platforms, we can extend the languages' grammars to incorporate specific aspects of the UI and native frameworks used in the development. For instance, in the case of Dart, the programming language grammar can be extended to have new eCST nodes defined by the Flutter framework. Taking into account all the new syntax, we can have nodes for threads, GUI elements, and mobile events. This means that our comparison algorithm for the eCST will have more information available, and therefore raise the precision of our algorithm.

4. Related Work

This section discusses existing strategies for clone detection in both the single-language and multi-language settings. We compare these strategies with our work.

Existing clone detection tools focus on a single programming language, and are not directly applicable to our case. There is a vast diversity in the target language to analyze; with prominence in C/C++, Java, JavaScript, and Python tools [Roy et al. 2009]. Existing tools aim for language independence so that clone detection is independent from tools' language, still, detection is possible for fragments of a single language [Cheng et al. 2017]. We note the approaches for single language clone detection are not easily transferable to the multi-language case, and therefore the algorithms and metrics are not comparable with OUT OF STEP.

C2D2 is one of the first approaches to detect clones across the .NET language family [Kraft et al. 2008]. Very similarly to our approach, C2D2 applies a hybrid method on tokens and ASTs for clone detection. However, contrary to our approach where any language may be supported, C2D2 may support only .Net languages which is a restriction caused by the intermediate representation chosen. A similar analysis of .NET languages uses a representation based on the Common Intermediate Language (CIL) [Al-Omari et al. 2012]. CIL is a stack-based machine-independent object-oriented

assembly language focused on .Net languages. Again, this tool differs from OUT OF STEP as it applies the Longest Common Sub-sequence (LCS) algorithm for detection, which can be restrictive in the detection of Type 2 and 3 clones.

LICCA [Vislavski et al. 2018] uses the original version of eCSTs to detect clones across different language types (*e.g.,* Java, Scheme). The eCST structure used in LICCA is the base for OUT OF STEP. As a consequence, this is the approach closest to our proposal. However, in OUT OF STEP we extend the generation of eCSTs to start directly from the PT and not use the AST. Moreover, we extend the definition of nodes in eCSTs (universal, advance, and stop nodes) to enable finer-grained analysis of tree sub-structures. Finally, while LICCA applies the LCS algorithm, our detection algorithm relies on tree mappings and broadcasting to map different node types, giving us more flexibility and possibilities to compare trees from across languages.

CLCSD [Zhang et al. 2020] is a hybrid framework for cross-language clone detection based on flowcharts as a means to abstract language constructs. Flowcharts in CLCSD have a tree-like structure similar to our eCSTs, standardizing different program structures. However, the standardization of flowcharts restricts the analysis for the base language grammar (Java and Python), not extensible to multiple different languages as OUT OF STEP. The closest approach to OUT OF STEP uses unified PTs generated from a language grammar linearized and pre-processed at the function level, comparing the linearized sequences [Nichols et al. 2019]. However, this approach differs from OUT OF STEP in the detection algorithm.

In addition, one of the main problems with existing clone detection tools is their lack of support [Walker et al. 2020]. We identified 20 clone detection tools from the literature, summarized in Table 2 with their maintainability state. Out of all the tools, 9 where downloadable, and only 3 of them could be installed and gave an output to a test file. Furthermore, most of the tools are available do not support the latest versions of the target programming languages. This means that to analyze code exploiting new features, the tools will return unexpected results.

Table 2. Existing clone detection tools

#	Name	Multi-language	Type	Runs
1	NICAD [Cordy and Roy 2011]	No	Textual	No
2	CCFinderX [Kamiya et al. 2002]	No	Lexical	No
3	Simian [Harris 2018]	No	Textual	No
4	Duploc [Gordon and Bannier 2021]	No	Textual	Yes
5	SourcererCC [Sajnani et al. 2016]	No	Lexical	No
6	iClones [Göde and Koschke 2009]	No	Lexical	No
7	PMD/CPD [pmd 2021]	No	Lexical	Yes
8	Deckard [Lingxiao et al. 2018]	No	Tree-based	No
9	Licca [Vislavski et al. 2018]	**Yes**	Hybrid	**Yes**

5. Conclusion and Future Work

Cross-language clone detection can enhance coed quality analysis, bug detection and feature divergence for polyglot software systems. Currently there are a small set of working tools for cross-language clone detection, mainly focused on classic languages (*e.g.,* Java, C++, Python). We identify a need for new algorithms and tools capable of performing

such task on modern languages. In this work, we propose an approach, with its accompanying functional tool, for the detection of Type 1, 2, and 3 clones across mobile programming languages. Our approach to cross-language clone detection, OUT OF STEP, uses a mixture of the abstract representation of programs based on their programming language grammars, and the textual code. For this purpose, we posit an extension to eCSTs that uses universal node types, where tree nodes carry token information. This helps us to explore the code and find the best corresponding clone matches.

The evaluation results confirm the main contribution of OUT OF STEP as an effective tool to detect clones across languages: (1) detecting clones for programming languages' main features with precision of over 79%, and (2) detecting clones with a precision of 65% to 84% for large mobile app code bases. Additionally, we posit a new corpus for cross-language clone detection, as no comprehensive data set exist to date, which is unfavorable for algorithm comparison. The corpus is used to contrast the divergence in application examples and target programming languages and application domains not covered by existing tools. Our corpus, evaluates basic language features, known domain algorithms, and equivalent mobile apps to maintain a common standard across languages.

OUT OF STEP is a first effective approach and tool for cross-language clone detection for mobile apps. While our results are promising, we identify further improvements for our approach. In the analysis of mobile apps, it is necessary to extend the eCST to include characteristics specific to mobile GUIs and frameworks. This will enable us to identify clones for GUI elements, threads, and mobile events. Another approach to consider is to investigate the comparison of the code not from the source, but once it is compiled to an intermediate or native representation, removing the layer of mobile frameworks. The main idea behind these improvements is to standardize eCSTs so that analyses do not diverge across languages, and therefore increase the precision of our algorithm. Additionally, we can introduce new tree transformations for fragments in which we can manipulate and reorder statements as to reduce the number of comparisons improving the complexity of the algorithm. Finally, we will extend the coverage of OUT OF STEP to include Swift (currently under development) to cover all languages for mobile platforms.

References

(2021). PMD: An extensible cross-language static code analyzer.

Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., and Maqbool, B. (2019). A Systematic Review on Code Clone Detection. *IEEE Access*, 7:86121–86144.

Al-Omari, F., Keivanloo, I., Roy, C. K., and Rilling, J. (2012). Detecting clones across microsoft. net programming languages. In *2012 19th Working Conference on Reverse Engineering*, pages 405–414. IEEE.

Budimac, Z., Rakić, G., and Savić, M. (2012). Ssqsa architecture. In *Balkan Conference in Informatics*, BCI'12, page 287–290, New York, NY, USA. ACM.

Cheng, X., Peng, Z., Jiang, L., Zhong, H., Yu, H., and Zhao, J. (2017). Clcminer: detecting cross-language clones without intermediates. *IEICE TRANSACTIONS on Information and Systems*, 100(2):273–284.

Cordy, J. R. and Roy, C. K. (2011). The NiCad Clone Detector. In *IEEE International Conference on Program Comprehension*, pages 219–220.

Göde, N. and Koschke, R. (2009). Incremental Clone Detection. In *European Conference on Software Maintenance and Reengineering*, SMR'09, pages 219–228.

Gordon, S. and Bannier, B. (2021). xsgordon/duplo-fork: C/C++/Java Duplicate Source Code Block Finder.

Harris, S. (2018). Simian - Similarity Analyser — Duplicate Code Detection for the Enterprise — Overview.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670.

Kraft, N. A., Bonds, B. W., and Smith, R. K. (2008). Cross-language clone detection. In *SEKE*, pages 54–59.

Lingxiao, J., Ghassan, M., Zhendong, S., and Stephane, G. (2018). skyhover/Deckard: Code clone detection; clone-related bug detection; sematic clone analysis.

Mondal, M., Roy, C. K., and Schneider, K. A. (2020). A survey on clone refactoring and tracking. *Journal of Systems and Software*, 159(110429):27.

Nichols, L., Emre, M., and Hardekopf, B. (2019). Structural and nominal cross-language clone detection. In Hähnle, R. and van der Aalst, W., editors, *Fundamental Approaches to Software Engineering*, FASE'19, pages 247–263. Springer International Publishing.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*, volume 1. 1 edition.

Patkar, N., Ghafari, M., Nierstrasz, O., and Hotomski, S. (2020). Caveats in eliciting mobile app requirements. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE'20, pages 180–189, New York, NY, USA. ACM.

Rakić, G. and Budimac, Z. (2013). Introducing enriched concrete syntax trees. In *Proceedings of the International Multiconference on Information Society*, pages 211–214.

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. (2016). SourcererCC: Scaling code clone detection to big-code. In *International Conference on Software Engineering*, pages 1157–1168. IEEE Computer Society.

Vislavski, T., Rakić, G., Cardozo, N., and Budimac, Z. (2018). Licca: A tool for cross-language clone detection. In *International Conference on Software Analysis, Evolution and Reengineering*, SANER'18, pages 512–516. IEEE.

Walker, A., Cerny, T., and Song, E. (2020). Open-source tools and benchmarks for code-clone detection. *ACM SIGAPP Applied Computing Review*, 19(4):28–39.

Zhang, F., Li, L., Liu, C., and Zeng, Q. (2020). Flow Chart Generation-Based Source Code Similarity Detection Using Process Mining. *Scientific Programming*, 2020.