DynAMICS: A tool-based method for the specification and dynamic detection of Android behavioural code smells

Dimitri Prestat, Naouel Moha, Roger Villemaire and Florent Avellaneda

Abstract—Code smells are the result of poor design choices within software systems that complexify source code and impede evolution and performance. Therefore, detecting code smells within software systems is an important priority to decrease technical debt. Furthermore, the emergence of mobile applications (apps) has brought new types of Android-specific code smells, which relate to limitations and constraints on resources like memory, performance and energy consumption. Among these Android-specific smells are those that describe inappropriate behaviour during the execution that may negatively impact software quality. Static analysis tools, however, show limitations for detecting these behavioural code smells and properly detecting behavioural code smells requires considering the dynamic behaviour of the apps. To dynamically detect behavioural code smells, we hence propose three contributions : (1) A method, the DYNAMICS method, a step-by-step method for the specification and dynamic detection of Android behavioural code smells; (2) A tool, the DYNAMICS tool, implementing this method on seven code smells; and (3) A validation of our approach on 538 apps from F-DROID with a comparison with the static analysis detection tools, ADOCTOR and PAPRIKA, from the literature. Our method consists of four steps: (1) the specification of the code smells, (2) the instrumentation of the app, (3) the execution of the apps, and (4) the detection of the behavioural code smells. Our results show that many instances of code smells that cannot be detected with static detection tools are indeed detected with our dynamic approach with an average precision of 92.8% and an average recall of 53.4%.

Index Terms—Android, code smells, detection, dynamic analysis, instrumentation, mobile apps, apps, behavioural

1 INTRODUCTION

In the last decades, the mobile apps market has grown tremendously and mobile apps have morphed from simple applications to rapidly evolving complex systems. For instance, in 2021 there were more than five million apps available in various app stores [1], with more than 230 billion downloads in 2021 [2]. Furthermore, to meet this increasing demand, mobile apps are developed at a rapid pace and are constantly evolving to meet new user requirements. However, these quick developments to solve bugs or add missing features in a constrained time frame may lead to poor design or implementation choices, also called *code smells* [3], which reinforce the technical debt.

Even if these mobile apps are mostly developed with Object-Oriented (OO) languages and many questions on OO code smells have already been addressed in the literature [4] [5], mobile apps bring new concerns, such as energy consumption, limited memory and limited performance. Due to these new concerns, the research community introduced new *Android-specific code smells*, describing them carefully to detect and correct them [3] [6] [7].

Some of these Android-specific code smells are considered as *behavioural* code smells. By defining a *behaviour* as a sequence of observable code events during execution, we define a *behavioural code smell* as *source code characteristics* inducing an inappropriate behaviour that may negatively impact software quality in terms of performance, energy consumption, memory. Here the source code characteristics refer to specific method calls/declarations or usage of specific code structures.

For example, the code smell Durable WakeLock (DW) manifests itself when the lock of a *WakeLock* is not released, causing battery drain. The *WakeLock* is the mechanism allowing an app to keep the device on. The DW code smell, therefore, describes the following inappropriate behaviour: A call to the *acquire* method is not followed by a call to the *release* method. In this case, the source code characteristics are the invocations of the *acquire* and *release* methods of the *WakeLock* class. While a static approach can be used to verify the presence of these two methods in the code, it does not allow to verify the behavioural aspect of the code smell. Indeed, it would be quite challenging to statically check that the two methods are called in the right order.

We have recently conducted an empirical study [8] on the effectiveness of Android behavioural code smells detection by the tools available in the literature, particularly ADOCTOR [7] and PAPRIKA [6]. That study shows that these static code smells detection tools show limitations in detecting behavioural code smells. The study, also argues that a dynamic approach could be a solution for the detection of behavioural code smells and urges the research community to introduce such tools.

In fact, we identify three behavioural code smells categories. The first one is characterised by the misuse of a method call or a sequence of method calls during the execution. The second is characterised by runtime issues, such as a long execution time of a method or an excessive

• *D. Prestat, R. Villemaire and F. Avellaneda are at Université du Québec à Montréal, Quebec, Canada.*

• *N. Moha is at École de Technologie Supérieure, Montréal, Canada*

Manuscript received...

use of the memory. Finally, the third is characterised by undesired data variations during execution, such as the size of a structure becoming excessively large. Each behavioural code smell belonging to these categories requires specific information that can be retrieved while the application is running [8]. A detection using dynamic analysis can easily provide these runtime information, unlike static analysis, which requires the static detection rules to be adapted to overcome the need of those runtime information.

This paper presents three contributions to address the lack of dynamic approaches for detecting behavioural code smells. First, we propose a method that contains all the necessary steps for the specification and detection of behavioural code smells. This is a major contribution, as all state-of-the-art works on Android code smells detection are solely static.

Second, we implement our method through a tool, named DYNAMICS (DYNamically Analysis of Mobile Instrumented Code Smells). DYNAMICS makes it possible to carry out all the steps automatically. DYNAMICS allows detecting code smells directly from the APK of an app. To accomplish this, from the code smells specification, our tool instruments the app, executes it to produce a trace and then detects the code smells according to this trace. As a result, DYNAMICS is a tool that allows the app's behaviour to be considered to ensure a more accurate detection of behavioural code smells.

Third, we validate DYNAMICS using precision and recall on an open-source dataset of 538 apps from F-DROID. Moreover, we compare it to two static analysis tools, ADOCTOR and PAPRIKA, for the detection of behavioural code smells from the literature. Our results show the effectiveness of our method for the detection of behavioural code smell, by pointing out that many instances of code smells that cannot be detected with static approaches are indeed detected with our tool.

This paper is organised as follows. Section 2 gives a background on behavioural code smells. Section 3 discusses the related work. Section 4 presents our method to detect behavioural code smells. DYNAMICS is presented in Section 5. The validation and the results of DYNAMICS are presented in Section 6. We conclude the paper in Section 7.

2 BEHAVIOURAL CODE SMELLS

Code smells have been initially introduced by Fowler [3] and Brown *et al.* [9], who describe them as symptoms of poor design or implementation choices made by developers during the development of a software system [3]. As the name indicates, these symptoms are *sniffable*, i.e. easily detectable in a static way. However, code smells identified in the literature can be challenging to ascertain, particularly for Android code smells. Nonetheless, we propose to focus on behavioural code smells in mobile apps for two main reasons. First, these code smells may hinder the software quality of mobile apps, specifically in terms of energy consumption, memory and performance. Secondly, existing research has not specifically addressed their detection.

We now present the seven code smells implemented in our tool. We follow the definitions given by [6], [7] and [10].

Durable WakeLock (DW): A *WakeLock* is the mechanism allowing an app to keep the device on in order to complete a task. However, when such task is completed, the lock should be released to reduce battery drain [11]. In Android, the class *PowerManager.WakeLock* is in charge of defining the methods to acquire and release the lock. If a method using an instance of the class *WakeLock* acquires the lock without calling the release, a smell is identified.
Code Characteristics: The call to the *acquire* and *release* methods of the *WakeLock* class.
Inappropriate Behaviour: A call to the *acquire* method is not followed by the call of the *release* method.

HashMap Usage (HMU): The Android framework provides *ArrayMap* and *SimpleArrayMap* as replacements from traditional Java *HashMap*. They are intended to be more memory-efficient and to trigger less garbage collection with no significant difference on operations performance for maps containing up to hundreds of values [12]. So, unless a complex map for a large set of objects is required, the use of *ArrayMap* should be preferred over the usage of *HashMap* in Android apps. Therefore, creating small *HashMap* instances is considered as a code smell [12], [13].
Code Characteristics: The call to a method from the *HashMap* / *ArrayMap* / *SimpleArrayMap* classes.
Inappropriate Behaviour: A *HashMap* structure is used for a small set of objects or *ArrayMap* / *SimpleArrayMap* structures are used for a large set of objects.

Heavy AsyncTask (HAS): In Android, the *AsyncTask* API allows developers to perform short background operations. However, three out of the four steps of *AsyncTask* are executed on the main UI thread and not in the background. Thus, these steps should not be time-consuming or use blocking operations to avoid: i) the GUI becoming unresponsive to user interactions or ii) the ANR dialog being shown. Thus, a class extending *AsyncTask* should never contain time-consuming or blocking *onPostExecute*, *onPreExecute*, or *onProgressUpdate* methods [14].
Code Characteristics: The implementation of *onPostExecute* / *onPreExecute* / *onProgressUpdate* methods within an *AsyncTask* class.
Inappropriate Behaviour: The *onPostExecute* / *onPreExecute* / *onProgressUpdate* methods are time-consuming or blocking.

Heavy BroadcastReceiver (HBR): Android apps can use a broadcast receiver to manage broadcast communications with the system or other apps. However, the *onReceive* method of *BroadcastReceiver* runs in the main UI thread. Thus, if this method contains time-consuming or blocking operations, it may also cause the app to freeze or to show an ANR dialog [15].
Code Characteristics: The implementation of the *onReceive* method within a *BroadcastReceiver* class.
Inappropriate Behaviour: The *onReceive* method is time-consuming or blocking.

Heavy Service Start (HSS): Services in Android can perform heavy operations in background. However, Android services run in the main thread of their hosting

process. By default, the service execution starts with a call to the *OnStartCommand* of the service, which runs in the main UI thread. Thus, the *OnStartCommand* should never contain time-consuming operations, otherwise, it may cause the app to freeze or to display an ANR (Application Not Responding) dialog [16]. Instead, when the service executes time-consuming or asynchronous operations, a new thread should be created by the method *OnStartCommand* to handle these operations outside the main UI thread.
Code Characteristics: The implementation of the *onStartCommand* method within a *Service* class.
Inappropriate Behaviour: The *onStartCommand* method is time-consuming or blocking.

Init OnDraw (IOD): *OnDraw* routines are responsible for updating the GUI of Android apps. These routines are invoked each time the GUI is refreshed (up to 60 times per second), and thus any extra computational work done in *OnDraw* is magnified by that frequency. Moreover, a high rate of memory allocations may lead to high memory consumption and numerous calls to garbage collection activities [17]. Thus, ideally, *OnDraw* routines should never contain *init* instructions to allocate memory (either new or calls to factory/constructor).
Code Characteristics: The implementation of the *onDraw* method within a *View* class.
Inappropriate Behaviour: The *onDraw* method is time-consuming or initialises objects.

No Low Memory Resolver (NLMR): When the Android system is running low on memory, the system calls the method *onLowMemory* of every running activity. This method is responsible for trimming the memory usage of the activity. If this method is not implemented by the activity, the Android system automatically kills the process of the activity to free memory. This may lead to an unexpected program termination. It is assumed that when the method *onLowMemory* is declared, it has to contain an event about the memory. This method is now deprecated and has been replaced by the *onTrimMemory* method [18] that allows to incrementally unload resources based on a parameter indicating the amount of trimming the app may like to perform. However, *onLowMemory* is still frequently used, for instance in our dataset, and this code smell has hence been retained for our tool. Furthermore, the detection method that we use can as well be applied with *onTrimMemory* and our tool could readily be extended to this method.
Code Characteristics: The implementation of the *onLowMemory* method within an *Activity* class.
Inappropriate Behaviour: The *onLowMemory* method does not reclaim memory when executed.

3 State of the Art

In this paper, we address (1) the “Detection of Code Smells” in (2) “Mobile Apps” using (3) “Dynamic Analysis”. Since to the best of our knowledge, no work has been done on these three aspects simultaneously, we present related work that considers two of these three aspects.

3.1 “Detection of Code Smells” in “Mobile Apps”

Reimann *et al.* [11] propose a catalogue of 30 quality smells dedicated to Android. These code smells originate mainly from the good and bad practices documented online in Android documentation or by developers reporting their experience on blogs. These code smells concern various aspects like implementation, user interfaces or database usage. These code smells are reported to have a negative impact on properties, such as efficiency, user experience or security. Two (DW and NLMR) of our studied code smells come from this catalogue.

Security smells [19] are another category of smells focused on the vulnerabilities in mobile apps. Ghafari *et al.* [19] identify 28 smells whose presence may indicate a security issue in a mobile app. These authors also developed a static analysis tool to study the prevalence of security smells. However, these code smells are not behavioural, as they concern the mere presence of an attribute in the manifest or the mere presence of a method call in the code independently of any induced inappropriate code behaviour.

Several tools are also available to detect Android code smells, and Rasool *et al.* [20] give a good overview of existing tools that can identify Android-specific code smells. For instance, Rasool *et al.* [20] own approach is able to recover 25 Android code smells by source code analysis and the computation of source code metrics. EARMO [21] furthermore reports an approach able to detect and correct code smells related to energy consumption within mobile apps. This approach, when used to correct these smells, is able to extend the battery life considerably. Multi-Objective Genetic Programming has also been used to detect Android smells [22]. This approach generates rules, which consist of a combination of quality metrics with threshold values to detect code smells. This method, therefore, takes as input a set of Android-specific code smell examples and finds the best set of rules to cover most of the expected Android code smells.

However, of the 19 different tools reported by Rasool *et al.* [20], if one removes those that are not available (prototypes, commercial or private tools), and those addressing non-behavioural code smells, all those that remain are extensions of aDoctor [7] or Paprika [6]. These two tools are therefore of prominent importance in Android behavioural code smells detection.

aDoctor [7] is a lightweight detection tool able to identify 15 Android-specific code smells. Paprika [6] is also a detection tool able to identify 17 Android code smells. Both aDoctor and Paprika are static analysis tools, and while aDoctor operates on the source code, Paprika processes the byte code. Iannone *et al.* [23] proposed a new version of aDoctor, which helps developers refactor the smells automatically. This extended version is open-source and available in Android Studio as a plugin published in the official store. Sniffer [24] is an open-source toolkit that tracks the full history of Android-specific code smells. However, Sniffer is not another detection tool because it relies on Paprika to detect Android code smells. In Sniffer, Paprika has been slightly modified to be able to analyse the source code directly instead of the byte code.

3.2 “Dynamic Analysis” used in “Mobile Apps”

Dynamic analysis of mobile apps has been used numerous times in related work. Usually, dynamic analysis is used to inspect the behaviour of malicious apps in order to detect malwares. For instance, ANDLANTIS [25], a highly scalable system capable of analysing 3,000 apps per hour dynamically, shows that it is possible to consider large-scale dynamic analysis to evaluate runtime behaviour and network traffic. Furthermore, DROIDTRACE [26] is a tool to study the malicious behaviour of malware. It uses PTRACE (Process Trace), a system call to observe and control the execution of another process. DROIDTRACE is in particular able to perform forward execution to trigger different dynamic loading behaviour. DROIDTRACE shares some similarities with the approach used in this paper in that both tools trace calls that occur during execution. However, DROIDTRACE is restricted to the dynamic load of libraries, while we consider any call that may be of interest for the considered code smell.

3.3 “Detection of code smells” via “Dynamic Analysis”

Although there is no literature on dynamic Android code smells detection as far as we know, there is some literature on the detection of code smells via dynamic analysis. For instance, JSNOSE [27] detects JavaScript code smells using dynamic analysis. However, the detection of code smells is done according to the values of software metrics and the dynamic aspects do not concern the behaviour of the app, but rather the computation of metrics and code coverage. The Feature Envy code smell, an OO code smell, has also been detected with a dynamic approach [28]. As in this paper, the Java code is instrumented and the program’s behaviour is then analysed during the execution. The paper also mentions that dynamic analysis could be advantageous for the detection of other types of code smells, however, detection is only performed on a single object-oriented code smell. Another difference with our approach is that that paper uses Aspect Oriented Programming to instrument the source code, while our approach instruments the bytecode (APK) and do not necessitate the mobile app’s source code.

Much research has focused on the detection of code smells in mobile apps and on dynamic analysis of mobile apps, which demonstrates the interest in these topics by the community. Few works exist on the detection of code smells via dynamic analysis, which shows that there are still many open research questions. However, there is a lack of approaches and tools to detect code smells, particularly behavioural code smells, using dynamic analysis in mobile apps. This, therefore, demonstrates the interest in the contributions of our paper to the state of the art.

4 DYNAMICS METHOD AND DYNAMICS TOOL

Various previous works show that many code smells can be detected by static analysis, including behavioural code smells. However, we will show that considering the behaviour of the app through dynamic analysis, these behavioural code smells can in fact be detected more accurately.

As a first contribution, we propose a method, the DYNAMICS method, covering all the essential steps, inputs and outputs for behavioural code smells detection considering the behaviour of the app. Our method is based on the four sequential steps depicted in Figure 1: *Specification*, *Processing*, *Execution* and *Detection*. The steps are summarised as follows:

- **Step 1. Specification:** For each code smell we determine, from its text-based description, the associated events and property. In this context, an *event* is a specific instruction or method call associated with the behaviour of the code smell. A *property* is a condition on events that determines, when it is satisfied, the presence of the code smell. Behavioural code smells are therefore represented by a series of properties on events that describe inappropriate behaviours of the mobile app. We, therefore, associate each code smell with a property to be checked such that as soon as this property is verified the code smell is detected.
- **Step 2. Processing:** The events associated to code smells specified in the first step are located within the app to enable them to be traced and used in subsequent steps.
- **Step 3. Execution:** The app containing the located events is executed according to scenarios and all encountered events are logged. Alternatively, it could also be possible to simulate the app’s behaviour instead of concretely executing it.
- **Step 4. Detection:** The detection of behavioural code smells is performed by analysing the sequence of events encountered during execution. Code smells whose associated property is satisfied will be detected.

The first step is performed once on all code smells, while the remaining steps must be reiterated for each new considered app.

As a second contribution, we have concretely implemented the DYNAMICS method in a tool, called the DYNAMICS tool. Figure 1 presents an overview of the four steps of the DYNAMICS tool: *Specification*, *Instrumentation*, *Execution*, *Detection*. Those steps are instances of the steps of the DYNAMICS method. It also emphasises the steps, inputs, and outputs specific to the DYNAMICS tool. The following items summarise the steps in the DYNAMICS tool:

- **Step 1. Specification:** As detailed in the DYNAMICS method, we define for each code smell a set of associated events and property. The events associated with a code smell represent source code characteristics that allow identifying this code smell in the APK. We, furthermore, express the properties as formal *Linear-time Temporal Logic* (LTL) properties whose satisfaction allows the code smell’s detection.
- **Step 2. Instrumentation:** For each code smell, we identify the source code characteristics represented by the associated events present in the APK. For each of these events, we instrument the app to insert instructions into the APK to generate specific log entries. This step was developed using the SOOT framework [29].

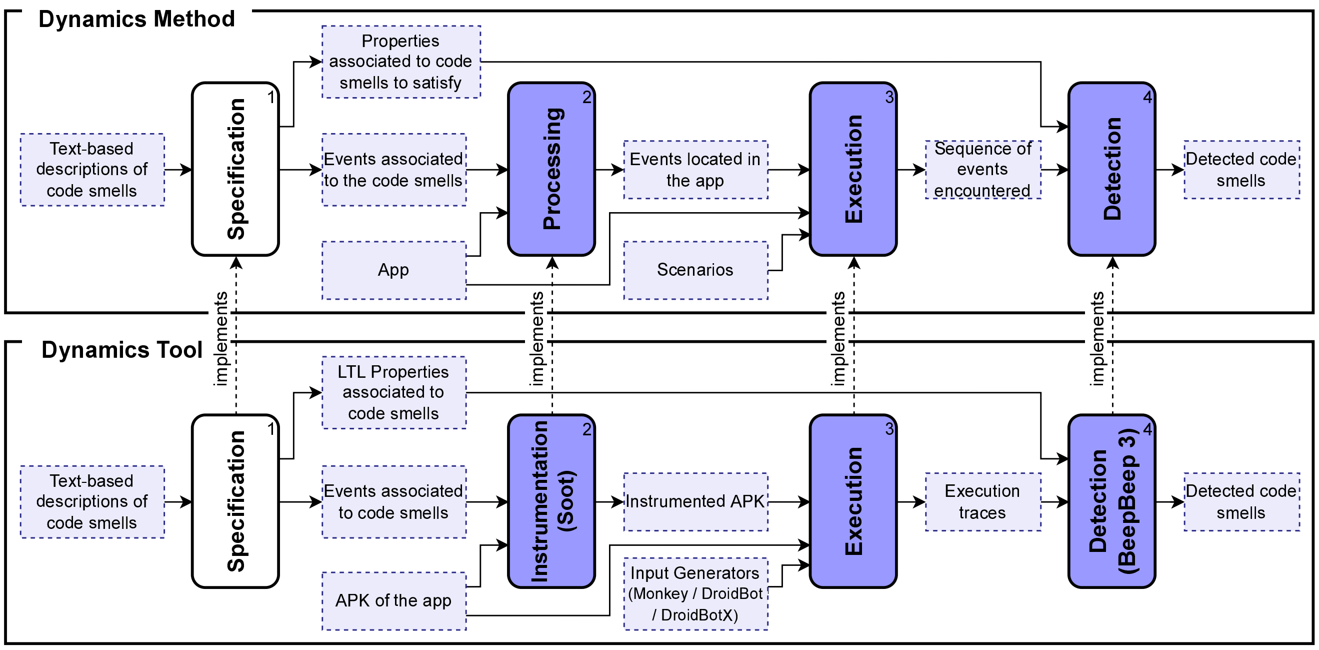
Dynamics Method
Text-based descriptions of code smells
Specification
1
Properties associated to code smells to satisfy
Events associated to the code smells
App
Processing
2
Events located in the app
Scenarios
Execution
3
Sequence of events encountered
Detection
4
Detected code smells
implements
implements
implements
implements
Dynamics Tool
Text-based descriptions of code smells
Specification
1
LTL Properties associated to code smells
Events associated to code smells
APK of the app
Instrumentation (Soot)
2
Instrumented APK
Input Generators (Monkey / DroidBot / DroidBotX)
Execution
3
Execution traces
Detection (BeepBeep 3)
4
Detected code smells

Fig. 1. Processes of DYNAMICS Method and DYNAMICS Tool. Boxes represent steps, arrows connect the inputs and outputs of each step described by dotted boxes. White boxes represent manual steps and filled boxes represent fully-automated steps.

- **Step 3. Execution:** We run the instrumented app automatically by emulation using input generators to monitor its events. As the events are encountered, multiple log entries are produced, and collectively these log entries form a trace. This step was developed using three input generators: MONKEYRUNNER [30], DROIDBOT [31] and DROIDBOTX [32]. An input generator is a program that simulates user interaction by generating the app's inputs.
- **Step 4. Detection:** From the complete trace representing all the events encountered in their precise order, we check if the LTL properties are satisfied or not. Each satisfied property leads to the detection of a code smell. This step was developed using the BEEPBEEP3 framework [33], a stream processing engine that allows the processing of log data.

The DYNAMICS tool, therefore, conducts an offline analysis on the generated traces since the analysis is done once the execution is completed. An online analysis might have been considered where the detection is done during the execution. However, this choice of an offline analysis was made to interfere as little as possible with the execution of the app and to reduce as much as possible the performance and memory footprint. This is of paramount importance since we consider some runtime and memory-related code smells.

5 The DYNAMICS Tool in Details

In the following, the four steps of the DYNAMICS tool are described using a common pattern: inputs, outputs, description, and implementation. The steps are, furthermore, illustrated by a running example using the DW code smell.

5.1 Step 1: Specification

Inputs: Text-based descriptions of mobile code smells from the literature.
Outputs: The events and the LTL property associated to each code smell. An *event* is a specific instruction or method call associated with the behaviour of the code smell. The *property* is a condition on events that determines, when it is satisfied, the presence of the code smell.

Description: We identify the events associated with code smells directly from the code smell's description. Based on these events, we specify the properties associated with code smells.

Each event is described by source code characteristics (specific method calls/declarations or usage of specific code structures (see Table 1)). Associated values are attached to an event depending on source code characteristics. These values can be timestamps, memory metrics or Java ID of objects and structures (see Table 1).

We define the properties using the Linear-time temporal logic LTL, a logic devised for expressing conditions on sequences. In our case, the sequences are those of the events associated with the code smell that appear in the execution trace. In a nutshell, *LTL* is a propositional, modal temporal logic first developed for the verification of reactive systems [34]. It augments propositional logic with the modalities F(eventually), G(always), X(next) and U(until) in support of expressing statements such as "A structure will *always* be small" or "A method will *eventually* be called". Such statements can be combined by means of logical connectors and nesting of modal operators to provide more involved properties. The syntax is natural and straightforward and, as a formal language, it has well-defined semantics and is therefore unambiguously interpretable.

TABLE 1
List of the events associated with code smells.

Code Smell	Events	Source Code Characteristics	Values	Property
DW	*Acquire*	*WakeLock.acquire*()	*WakeLock* ID	Formula for a *WakeLock* of ID x: $\boldsymbol{F}(acquire(x)\wedge$ $\neg\ (X\ (\neg acquire(x)\ U\ release(x))))$
	Release	*WakeLock.release*()	*WakeLock* ID	

HMU	*Instantiation*	map = new *HashMap*
		map = new *ArrayMap*
		map = new *SimpleArrayMap*
			Structure ID
			Size
			Type
				Formula for a map of ID x: $\boldsymbol{F}((type(x) = SimpleArrayMap$ $\vee type(x) = ArrayMap) \wedge size(x) \geq 500)$ $\vee \boldsymbol{G}(type(x) = HashMap \rightarrow size(x) \leq 500)$

	Addition	map.*put*(...) map.*putAll*(...)	Structure ID Size Type	
	Deletion	map.*remove*(...)	Structure ID Size Type	
	Clear	map.*clear*()	Structure ID Size Type	
HAS/HBR/HSS	*Begin*	Beginning of the method *onPreExecute* / *onPostExecute* / *onProgressUpdate* / *onReceive* / *onStartCommand*	Method call ID Timestamp	Formula for a method call of id x: $\boldsymbol{F}(begin(x) \wedge (X\ (\neg begin(x)\ U\ (end(x)$ $\wedge(endTime(x) - beginTime(x) > 100)))))$
	End	End of the method *onPreExecute* / *onPostExecute* / *onProgressUpdate* / *onReceive* / *onStartCommand*	Method call ID Timestamp	
IOD	*Begin*	Beginning of the method *onDraw*	Method call ID Timestamp	Formula for a method call of id x: $\boldsymbol{F}(begin(x) \wedge (X(\ \neg begin(x)\ U\ (end(x)$ $\wedge(endTime(x) - beginTime(x) > 1/60))))) \ \vee$ $\boldsymbol{F}(begin(x) \wedge \neg((\neg new)\ U\ end(x)))$
	End	End of the method *onDraw*	Method call ID Timestamp	
	New	Instantiation (<init>) in the *onDraw* method	Method call ID	
NLMR	*Begin*	Beginning of the method *onLowMemory* / *onTrimMemory*	Method call ID Memory state	Formula for a method call of id x: $\boldsymbol{F}(begin(x) \wedge (X\ \neg begin(x)\ U\ (end(x)\wedge$ $(memoryEnd(x) - memoryBegin(x) < 1024))))$
	End	End of the method *onLowMemory* / *onTrimMemory*	Method call ID Memory state	

Each property is, furthermore, parameterised by a variable x. This variable x refers either to the instance of an object or to the call of a method. For example, for the DW code smell, the *WakeLock* of ID x will have the property φ_x.

In the following properties, there is a frequent use of the Until (U) operator. When two events, say $acquire$ and $release$, occur alternately, we are usually interested, for an $acquire$ event, in its corresponding $release$ event. This $release$ event obviously occurs after the $acquire$, but, furthermore, there are no additional $acquire$ until this $release$ occurs. This is formally expressed by the condition $X(\neg acquire\ U\ release)$. This expresses that, starting at the next event of the sequence, there is no $acquire$ event until a $release$ event occurs. We will use this formulation for $acquire/release$, but also for the $begin/end$ events.

Implementation: There is no implementation for this step. The specification, i.e., the definitions of events and property, is defined manually according to the description of the code smell.

Running example:
DW:
Events: The events are triggered by calls to the methods *acquire* and *release* of the *WakeLock* class. The associated value is an integer indicating the id of the *WakeLock* instance. The events *Acquire* and *Release* are associated to a unique *WakeLock* instance.
Property: A code smell is detected each time an *acquire* on a *WakeLock* is encountered but there is no *release* associated.
"Eventually, the *WakeLock* x is acquired and it is not the case that its corresponding release occurs."
LTL Formula for a *WakeLock* of ID x:
$\varphi_x = \boldsymbol{F}(acquire(x) \wedge \neg\ (X\ (\neg acquire(x)\ U\ release(x))))$

5.1.1 *Specification of the other code smells*

We now detail, for each code smell detected by the DYNAMICS tool, their events and their property.
HMU:
Events: The events are triggered by calls to the methods of the classes *HashMap*, *SimpleArrayMap* and *ArrayMap* influencing the size of the structures: *new*, *put*, *putAll*, *remove*, *clear*. The values associated to each event are the type of structure, i.e., its class, its actual size and its id.
Property: A code smell is detected if a *SimpleArrayMap/ArrayMap* reaches a large size or if a *HashMap* never reaches a large size. Structures of up to several hundred elements are considered large [12]. Each set of values whose size exceeds or is lower than 500 depending on the type thus identifies a smell.
"Eventually (i.e., at some point in the sequence), the

structure is either a *SimpleArrayMap* or an *ArrayMap* and its size is greater or equal to 500."
$\varphi_{1x} = \boldsymbol{F}((type(x) = SimpleArrayMap \vee type(x) = ArrayMap) \wedge size(x) \geq 500)$
"Globally (i.e., for all events in the sequence), if the structure is a *HashMap*, its size is less or equal to 500."
$\varphi_{2x} = \boldsymbol{G}(type(x) = HashMap \rightarrow size(x) \leq 500)$
Since there are two cases to check, the LTL Formula φ_x for the structure x is the disjunction (logical "or") of the two previous LTL formulas $\varphi_x = \varphi_{1x} \vee \varphi_{2x}$.

HAS/HBR/HSS:
Events: The events are triggered by calls to methods specific to the code smell: *onPostExecute*, *onPreExecute* and *onProgressUpdate* for HAS, *onReceive* for HBR, *onStartCommand* for HSS. However, for all methods, there is a *Begin* and *End* events, respectively at the beginning and at the end of the method call. Each event is associated with a timestamp and the ID of the method call.
Property: A code smell is detected each time a method associated with HAS, HSS or HBR is encountered with an execution time greater than 100ms, which corresponds to the difference between the timestamps of the *Begin* and the *End* events.

"Eventually, the beginning of the method call x is reached and at the end of this method call, the elapsed time is greater than 100ms."
LTL Formula for a method call of id x:
$\varphi_x = \boldsymbol{F}(begin(x) \wedge (X \ (\neg begin(x) \ U \ (end(x) \wedge (endTime(x) - beginTime(x) > 100)))))$

IOD:
Events: The events are triggered by calls to the method *onDraw*. Here again, there are *Begin* and *End* events at the beginning and at the end of the method call. There is, furthermore, an additional *New* event triggered by the instantiation of new objects. Each event is associated with the ID of the method call. The *Begin* and *End* events are associated with a timestamp value.
Property: A code smell is detected each time an *onDraw* method is encountered with an execution time greater than 1/60 of a second, or if we encounter an instantiation. The difference between the timestamp values of the *End* and the *Begin* events is used to compute the execution time of the method.
"Eventually, the beginning of the method call x is reached and at the end of this method call, the elapsed time is greater than 1/60s, or the beginning of the method call x is reached and it is not the case that there is no instantiation before the call ends."
LTL Formula for a method call of id x:

$$\begin{aligned}\varphi_x = &\boldsymbol{F}(begin(x) \wedge (X(\neg begin(x) \ U \ (end(x) \\ &\wedge (endTime(x) - beginTime(x) > 1/60s))))) \ \vee \\ &\boldsymbol{F}(begin(x) \wedge \neg((\neg new) \ U \ end(x)))\end{aligned}$$

NLMR:
Events: The events are triggered by the execution of the methods *onLowMemory* and *onTrimMemory*. As before, there are *Begin* and *End* events at the beginning and end of the method. The *Begin* and *End* events are associated with the amount of memory used at that moment and the ID of the method call.
Property: A code smell is detected when an *onLowMemory* or *onTrimMemory* method is encountered and the released memory during this method call is inferior to 1024KB. The difference between the memory values of the *Begin* and *End* events is used to compute the amount of memory freed during the execution of the method. It is also detected if an *Activity* is present and do not define an *onLowMemory* or *onTrimMemory* method. This second part of the detection rule is detected statically and is not processed by the LTL formula.
"Eventually, the beginning of the method call x is reached and at the corresponding end the memory released during the call is inferior to 1024KB."
LTL Formula for a method call of id x:
$\varphi_x = \boldsymbol{F}(begin(x) \wedge (X \ \neg begin(x) \ U \ (end(x) \wedge (memoryEnd(x) - memoryBegin(x) < 1024))))$

5.2 Step 2: Instrumentation

Inputs: The APK to instrument and the events associated to the code smells.
Outputs: The instrumented APK with logging instructions.

Description: In this step, the objective is to instrument the APK of a mobile app by adding logging instructions. Concretely, the instrumentation consists in identifying the relevant events located within the APK and inserting an instruction that will produce a specific log entry for each event. Thus, when the instrumented app is executed, a trace composed of a sequence of log entries will be generated. In our context, a log entry is a tuple (location, id, event, values) where :

- *location* is the package, class and method names where the event occurs;
- *id* is a sequential identifier that distinguishes several occurrences of the same event in the same method of the same class;
- *event* is the keyword associated to an event. The events are listed in Table 1;
- *values* contains the values used for the LTL properties of the code smells detection. The values may be found in Table 1;

Furthermore, the instrumentation differs according to the category the code smell belongs to:

- Code smells regarding the misuse of a method call or a sequence of method calls. The instrumentation is done by inserting an instruction for each call of the concerned methods. A single code smell in this paper belongs to this category: *DW*.
- Code smells regarding runtime issues within a method. For these code smells the instrumentation is done by inserting an instruction at the beginning and at the end of the declaration of the concerned method. Five code smells in this paper belong to this category: *HAS*, *HSS*, *HBR*, *IOD* and *NLMR*. For these code smells, the associated events contain a method

call ID. This method call ID is determined from the code location and determines the id in the log tuple.

- Code smells regarding undesired data variations of an object during execution. The instrumentation is done by inserting an instruction for each call of a method that impacts the data of the concerned objects. A single code smell in this paper belongs to this category: *HMU*.

Implementation: This step requires a static analysis in order to instrument the code. To achieve this, we have developed a Java module, in Step 2 of the DYNAMICS tool, using the SOOT framework [29] and its DEXPLER module [35] to analyse APK artefacts. APK files are ZIP archives containing various information, such as the AndroidManifest.xml manifest defining all the metadata of the app and a .dex file containing all the classes of the app compiled into a dex format file [36]. The dex bytecode is register-based, which means that translating it into Java or intermediate languages implies an important loss of information. This is due to the fact that the type and name of local variables may be harder to retrieve, and that the branches (for, while, if, ...) are replaced by goto. SOOT converts the bytecode of APKs into a SOOT internal representation, which is similar to the Java language. DYNAMICS goes through the internal representation of SOOT for each class, each method and each instruction of the body of these methods. These instructions are converted into Jimple, a simplified version of Java source code. These instructions are analysed to detect if an event is present. If such an event is present, instructions allowing the generation of the log entries are inserted directly after the event in the APK. Finally, a small but necessary task, the generated APK must be signed in order to be executed in the next step. We use JARSIGNER [37] to perform this.

Analysing the bytecode directly rather than the source code has both advantages and disadvantages. For instance, we do not need to have access to the source code nor do we need to perform any compilation steps. On the other hand, we cannot know the original line number associated with the java instruction.

```
PowerManager.WakeLock completeWakeLock;
(...)
completeWakeLock.acquire();
```

Fig. 2. Java instruction example.

```
lambdaapp.LockManager.java$lock:0:dwacq:145
```

Fig. 3. Log output associated with the Java instruction.

Running example: Figure 2 shows a call to the *WakeLock*'s method *acquire* that locks the *WakeLock*. This is an *Acquire* event for the DW code smell. In this case, the instrumented app will output the log entry depicted in Figure 3. This entry shows the method name *lock*, the class name *LockManager* and the package name *lambdaapp* where this call occurs. It also shows that this is the first event of this type in the method thanks to the id 0. Finally, it indicates that it is an *Acquire* event thanks to the *dwacq* keyword and that it has operated on the structure of id 145.

5.3 Step 3: Execution

Inputs: The instrumented APK and the input generators.
Outputs: The execution traces obtained after running the instrumented app with the input generation tools.

Description: This step consists in executing the instrumented app installed in a real or virtual device using different input generators with different input configurations (such as allowed execution time) to produce execution traces. An input generator is a program that simulates user interaction by generating app's inputs, for example cliking a specific button, entering a text in an input text or going backward. This is an important step of the DYNAMICS tool since these traces will allow the detection of the code smells. Each time the execution reaches a logging instruction, a specific log entry will be produced. The execution trace is composed of all these log entries. Thus, the trace is composed of all the events associated with the code smells encountered during execution.

The challenge is naturally to reach a good coverage of the instrumented code at runtime to capture as many events as possible in order to potentially detect as many code smells as possible.

Implementation: The execution can either be done manually by an already provided execution scenario or automatically using an input generator. We chose to use input generation tools to automate this part. We compared the following tools: DYNODROID [38], DROIDUTAN [39], DROIDBOT [31], DROIDBOTX [32], HUMANOID [40] and MONKEYRUNNER [30]. After some preliminary experiments with these tools on the same set of mobile apps, comparing their ability to trigger the events detected by our DYNAMICS tool, we identified the following three input generators as the most promising:

- **MONKEYRUNNER**: This is a tool that performs random events on the user interface of the app. It is a free random test tool included in the Android SDK. This tool emulates a user interacting with an app, generating and injecting pseudo-random actions, for example, clicks, swipes, or system events into the app's event input stream.
- **DROIDBOT**: This tool is based on MONKEYRUNNER. DROIDBOT is an open-source testing tool that uses a model-based exploration strategy under a black box approach. It also allows users to customise their test scripts using the generated state transition model.
- **DROIDBOTX**: This tool is an extension of DROIDBOT that generates random actions based on the Q-Learning technique [41]. This approach systematically selects input events and guides exploration to expose the functionality of an app under test to maximise the coverage of instructions, methods and activities by minimising redundant input events.

We use these three input generation tools on an emulator to generate the traces that will be used for detection. We used the *adb* [42] command-line tool to install the APK. We also used the *logcat* [43] command-line tool to retrieve the log of the device, which contains the generated execution traces. We filter those logs to select only the logs that match our log format depicted in Section 5.2. Otherwise, there are no filtering on the sequences generated.

```
lambdaapp.LockManager.java$lock:0:dwacq:145
lambdaapp.LockManager.java$lock:1:dwacq:191
lambdaapp.LockManager.java$lock:2:dwacq:143
lambdaapp.LockManager.java$lock:0:dwrel:145
```

Fig. 4. Excerpt of an execution trace.

Running Example: Figure 4 is an excerpt of a trace generated during an execution for the detection of the DW code smell. The excerpt comes from the running example given in Step 2. There are three *WakeLocks* of different identities as shown in the trace. Only one of the *WakeLocks* is acquired and then released, the two others are acquired but never released. Therefore, the first one, of id 145, does not signal the presence of a code smell whereas the two others, of ids 191 and 143, do.

5.4 Step 4: Detection

Inputs: A set of execution traces and the LTL properties associated to the code smells.
Outputs: The detected code smells.

Description: This step consists in analysing the resulting execution traces using the technique of runtime monitoring to identify the code smells that have occurred. Concretely, runtime monitoring consists in analysing the sequences of events within execution traces and checking the LTL properties on these sequences.

As seen in the specification step, the properties are parameterised with a variable x. This variable x refers either to the instance of an object or the call of a method. Only events associated with this instance or this method call must be used to check the property. This is necessary so that properties are not verified using events from different objects or method calls. For example, in order to check the DW property one must not use the *acquire* event of a *WakeLock* x_1 with the *release* event of a different *WakeLock* x_2. Only events concerning the *WakeLock* of ID x must be used in verifying property φ_x.

Implementation: For this step, we have implemented a Java module that performs runtime monitoring in our tool using the BEEPBEEP 3 framework [33]. BEEPBEEP 3 is a stream processing engine that allows the processing of log data. It allows multiple processing on the traces, including verification of LTL formulas. We thus define a chain of BEEPBEEP 3 processes allowing us to check if it is indeed an event trace linked to a code smell. We specify a branch per code smell to check the associated LTL property. When the whole trace is processed, we obtain the detected code smells.

Precisely, the chain can be divided into three steps: (1) keep only DYNAMICS specific input lines; (2) keep only input lines for a specific code smell; (3) check this specific code smell's LTL property using the input lines' events information in order to determine the presence of code smells.

BEEPBEEP 3 also provides the possibility to slice the trace according to a parameter. This allows us to check an LTL formula for each element of a code smell. For example, it allows us to check an LTL property for each *HashMap* of different IDs for the HMU code smell. It also allows us to check one LTL property for each different *onDraw* method call encountered for the IOD code smell.

Running Example:
Figure 5 shows the processor chain required to detect the DW code smell from an execution trace. BEEPBEEP 3 consists of processors linked together by pipes producing an output according to an input.

In the figure a processor is represented by a square box, with a pictogram representing the type of computation it executes on events. On the sides of this box are one or more "pipes" representing its inputs and outputs. Input pipes are indicated with a red, inward-pointing triangle, while output pipes are represented by a green, outward-pointing triangle. The colour of the pipe indicates the nature of the data flowing in the chain.

The processors 1 to 3 correspond to the first step : keep only DYNAMICS specific input lines. The processors 4 to 11 correspond to the second step : keep only input lines for a specific code smell. The processors 12 to 19 correspond to the third step : check this specific code smell's LTL property using the input lines' events information in order to determine the presence of code smells.

First of all, processor 1 (*ReadLines*) reads the data line by line from a file, in our case an execution trace of an app. It is therefore the lines of the file that will run through the pipes one by one. Processor 2 (*ToString*) then converts it to a String while processor 3 (*FindPattern*) uses a regular expression to keep only lines of the form *location:id:event:values*. Processor 4 is a *Fork* that duplicates the stream into multiple processors. Processors 5 (QueueSource) and 6 (function Contains Strings) verify whether the stream contains the *Acquire* event while, in parallel, processors 7 and 8 verify if the stream contains the *Release* event. Processor 9 (Function Or) then verifies if the stream contains an *Acquire* event or *Release* event. Processor 10 (Filter) will continue to transmit the stream only if the input Boolean is true, which, in our case, means that the stream continues in this branch only if it contains an *Acquire* or *Release* event, the two events related to the DW code smell. Processor 11 (Function split string) splits the string in an array, using the double point as a separator and processor 13 (Slice) splits the events from the stream into multiple substreams according to a parameter, in our case the value of the log entry, the WakeLock ID. We, therefore, obtain as many substreams as *WakeLock* IDs encountered.

The output is a map containing the Id of the *Wakelock* as a key, and a Boolean (true or false) indicating whether the code smell is present or not as a value. The processors 15 and 16 (Function NthElement) select the first and fourth elements of the array, i.e., the location and the Id of the

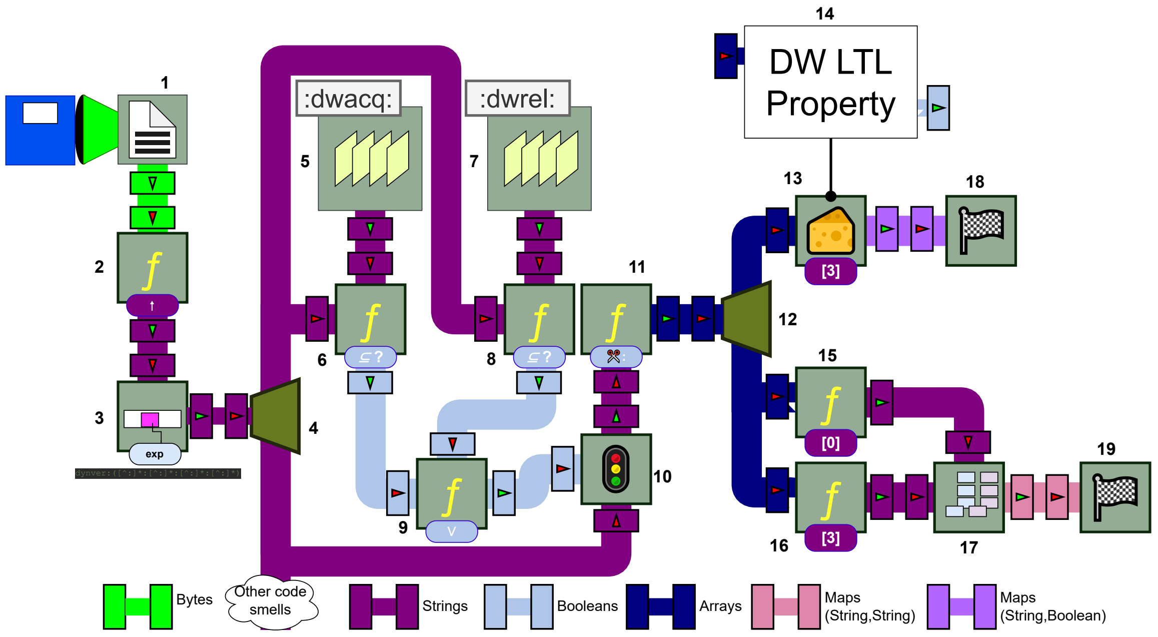

Fig. 5. Excerpt of the BEEPBEEP processor chain for the detection of the DW code smell.

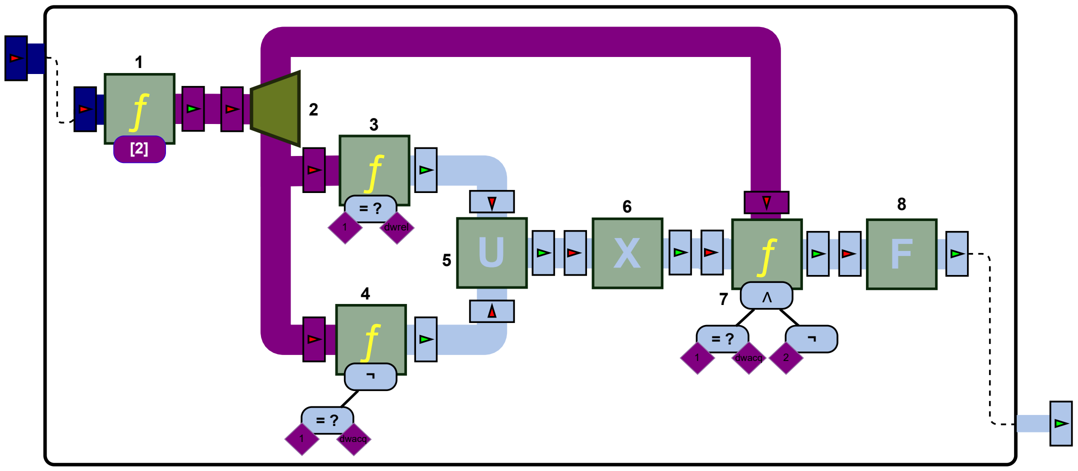

Fig. 6. The BEEPBEEP processor chain for the LTL property for the detection of the DW code smell.

WakeLock that are then gathered in a map by processor 17 (PutInto (Maps)).

The processors 18 and 19 (KeepLast) are special processors that simply return the last events, in our case the filled maps. So at the end of this stream, we get two maps, one linking locations to IDs, and one linking IDs to the presence of code smell.

Figure 6 shows the substream representing the LTL formula for the detection of DW code smell, as depicted in the "processor" 14 of the Figure 5. Here the formula to check is $\varphi_x = \boldsymbol{F}(acquire(x) \wedge \neg\,(X\ (\neg acquire(x)\ U\ release(x))))$. The processor 1 select the second element of the input, here the name of the event encountered and the processor 2 dispatch the event. The processor 3 corresponds to $release(x)$ and the processor 4 corresponds to $\neg acquire(x)$. The processor 5 takes the processors 3 and 4 as inputs to represent $(\neg acquire(x)\ U\ release(x))$. The processor 6 takes the processor 5 as input to represent $(X\ (\neg acquire(x)\ U\ release(x)))$. The processor 7 is the disjunction that takes the processor 6 and an *acquire* event from the processor 2, it then represents $(acquire(x) \wedge \neg\,(X\ (\neg acquire(x)\ U\ release(x))))$. Finally, we apply the processor 8 to add the Eventually temporal operator $\boldsymbol{F}$ to have the LTL property for DW code smell.

5.5 Discussion

When specific thresholds are used, the threshold values that we use come either from the references from which the code smells originate, such as for HMU or IOD or when the reference only provides a qualitative description, such as "time-consuming" or "reclaiming memory" for HAS, HSS, HBR and NLMR. For the latter case, we use a value that to the best of our knowledge matches this description taking into consideration execution time and memory footprint encountered during our tool's execution. The values for the thresholds may be subjective and open to discussion, but our method is generally applicable and our tool can easily be modified to accommodate alternative values. The threshold values used in DYNAMICS can be found in the definition of the LTL properties of the code smells. For example, 500 elements for HMU, 100 ms for HAS/HSS/HBR, 1/60s for IOD, 1024KB for NLMR.

The DYNAMICS tool is also extensible for new code smells. To detect a new code smell, it suffices to give the events associated and create an LTL formula for this code smell's property, as described in Step 1 Specification. The other steps can be easily applied to these new code smells.

6 VALIDATION

In this section, we present the study that aims to validate our proposed tool, DYNAMICS. We follow a mixed-method methodology through quantitative and qualitative data collection and analysis.

6.1 Research Questions

We aim to respond to the following three research questions:

- **RQ_1: Does DYNAMICS allows the detection of behavioural code smells ?** This question investigates the effectiveness of DYNAMICS for the detection of behavioural code smells. We attempt to identify the extent to which DYNAMICS return positives in the detection of behavioural code smells.

- **RQ_2: Does DYNAMICS performs better in terms of precision and recall than the state-of-the-art tools ?** This question investigates the good precision and recall of DYNAMICS on behavioural code smells through more adapted detection methods.

- **RQ_3: Are there instances detected by the dynamic analysis that could not have been detected statically ?** Due to the DYNAMICS detection rules based on dynamic analysis, we may detect instances of code smells that only occur during runtime and are therefore unlikely to be detected by static analysis.

6.2 Subjects of the Validation

The validation is conducted primarily on our tool, DYNAMICS. Then, we detect code smells on the same apps using two other tools, ADOCTOR and PAPRIKA to compare the results. These two tools are easy to use and open-source. These tools use static detection techniques and are fully automatic.

In a previous paper [8] we considered these two tools, PAPRIKA and ADOCTOR. We showed that they were representative of state-of-the-art tools. A systematic procedure was conducted to reduce the list of 19 state-of-the-art tools to these two tools. PAPRIKA was taken from [44] and ADOCTOR was taken from [45].

We use DYNAMICS to detect seven Android-specific behavioural code smells: Durable WakeLock (DW), HashMapUsage (HMU), Heavy AsyncTask (HAS), Heavy BroadcastReceiver (HBR), Heavy Service Start (HSS), Init OnDraw (IOD), No Low Memory Resolver (NLMR). None of the other tools detects exactly the same code smells and covers all of them. The aim is to compare the other tools with DYNAMICS, not to compare them with each other. PAPRIKA is able to detect HMU, HAS, HBR, HSS, IOD and NLMR whereas ADOCTOR is able to detect DW and NLMR. The definition of these code smells is provided in Section 2. The seven behavioural code smells detected by DYNAMICS are selected according to their depreciation and representativeness within the three behavioural categories, as specified in the empirical study on behavioural code smells [8].

6.3 Objects of the Validation

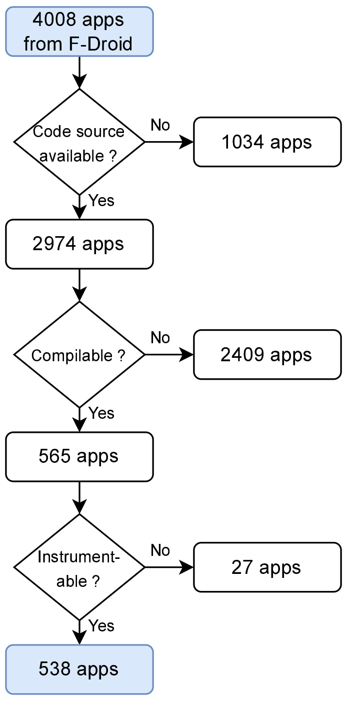

Fig. 7. Flowchart of the selection of the apps.

The study is based on 538 apps collected from F-DROID[1]. This dataset consists of real open-source apps from F-DROID published on GITHUB. F-DROID provides a dataset of real apps that are neither dummy apps, templates, nor libraries. To ensure that the apps and their associated source code are retrieved, we make sure that the apps come from GITHUB, are still available, and can be built and instrumented. The apps are selected in the following way, as depicted in Figure 7.

F-DROID provided 4008 apps. Of the 4008 apps, 3034 come from GITHUB, but only 2974 are still available on GITHUB. All apps are built using the build automation scripts provided by the developers on F-DROID. 565 apps (out of 2974) could be built, and the rest of the apps (2409 = 2974 - 565) could not due mainly to non-functional scripts or outdated elements. Finally, 538 (out of the 565 apps built) could be instrumented while the rest of the apps (27 = 565 - 538) could not mainly due to too old APIs and specific missing libraries.

We provide in [46] the complete list of our artefacts, which include the list of APKs, APKs, execution traces,

1. https://f-droid.org/

detected code smells for each tool, compiled results and our tool.

6.4 Validation Process

The process of the study as illustrated in Figure 8 consists of five main steps described in the following.
Step 1. Detection with the tools. We use ADOCTOR, DYNAMICS and PAPRIKA on the dataset of 538 apps to detect the seven behavioural code smells, hence retrieving the detected code smells (i.e., positive instances).
Step 2. Identification of potential code smells. In parallel, we identify the potential code smells across the 538 apps. These potential code smells are all the classes possibly affected by a code smell conforming to its definition. The potential code smells classes are identified during the instrumentation step of DYNAMICS (see Figure 1), in which the events associated with the potential code smells are located in the source code. The potential code smells classes are therefore identified as follows:

- **DW:** Unique classes containing the *Acquire* event. There are 45 *Acquire* events in 40 unique classes.
- **HMU:** Unique classes containing the HMU *Instantiation* event. There are 1559 HMU *Instantiation* events in 836 unique classes.
- **HBR:** Unique classes containing the *onReceive* method (*Begin*/*End* Events). There are 719 unique classes.
- **HSS:** Unique classes containing the *onStartCommand* method (*Begin*/*End* Events). There are 134 unique classes.
- **HAS:** Unique classes containing the *onPreExecute* / *onPostExecute* / *onProgressUpdate* methods (*Begin*/*End* Events). There are 703 methods in 509 unique classes.
- **IOD:** Unique classes containing the *onDraw* method (*Begin*/*End* Events). There are 130 unique classes.
- **NLMR:** Unique *Activity* classes. There are 18 such classes containing the *onLowMemory*/*onTrimMemory* methods (*Begin*/*End* Events) and 2002 *Activity* classes without these methods, for a total of 2020 unique classes.

The potential code smells are also shown in the artefacts [46].
Step 3. Filter undetected. For each tool, we filter the potential code smells to only retrieve the undetected potential code smells. The undetected potential code smells are the potential code smells that have not been detected by the tools, and are therefore not detected code smells. From all the classes containing a potential code smell retrieved in Step 2, we subtract all the classes detecting with a code smell from Step 1. For example, the undetected potential code smells from DYNAMICS for the HMU code smell are all classes potentially having the HMU code smell from Step 2 minus the classes detected by DYNAMICS as having the HMU code smell in Step 1. These are shown in the artefacts [46].
Steps 4 and 5. Sampling. To consider statistically significant samples for detected code smells in Step 4 and undetected potential code smells in Step 5, we rely on stratified samples.

TABLE 2
True/False Positive/Negative instances.

	Validated instances	
	Positives (V)	**Negatives (V^c)**
Detected code smells (D) (positive instances)	True Positive (TP) $D \cap V$	False Positive (FP) $D \cap V^c$
Undetected potential code smells (D^c) (negative instances)	False Negative (FN) $D^c \cap V$	True Negative (TN) $D^c \cap V^c$

These stratified samples ensure that the proportion of each code smell is preserved in the sample. The sample for each code smell consists of the detected code smells fetched by Step 1 and this method represents a 95% statistically significant stratified sample with a 10% confidence interval. Stratified sampling is furthermore always done on the accessible code smells, i.e., those that can be found in the source code. This is necessary since in order to validate the tools' results, in Steps 6 and 7, the source code must be manually inspected. The source code is however not required by our tool but used only during these two manual validation steps. Some code smells are however not accessible due to obfuscation, as PAPRIKA and DYNAMICS analyse the APK directly. This is also potentially a consequence of Kotlin, which generates functions at compile time. For instance, some of the generated functions contain *HashMap*, which results in HMU code smells not accessible to PAPRIKA nor DYNAMICS. It is also due to external libraries that are returned in the results, as is the case for some of the code smells detected by PAPRIKA.

From here, we can now distinguish the four types of instances shown in Table 2 that are determined in the following steps.
Step 6. Manual analysis. We analyse the positive sample manually to determine which are true positives and which are false positives. The true positives are the instances detected by the tools and validated manually in conformance with their code smells definition: $TP = D \cap V$. The false positives are the instances detected by the tools but validated manually not in conformance with the code smells definition: $FP = D \cap V^c$. The sample is manually validated by four PhD students with experience in code smells and mobile apps. Each sampled instance was validated by at least two people. In the few cases where there were discrepancies, we revisited the case to reach a consensus.
Step 7. Manual analysis. Similarly, we analyse manually the undetected potential code smells to determine which are true negatives and which are false negatives. The true negatives are the instances not detected by the tools and validated manually not in conformance to the definition: $TN = D^c \cap V^c$. The false negatives are the instances not detected by the tools but validated manually to be in conformance with the code smells definition: $FN = D^c \cap V$. The sample is manually validated by four PhD students with experience in code smells and mobile apps.
Step 8. Validation. We calculate the precision using the positive sample. Precision is the proportion of true positives among the positives (here, detected code smells) $\frac{|TP|}{|D|}$. Then,

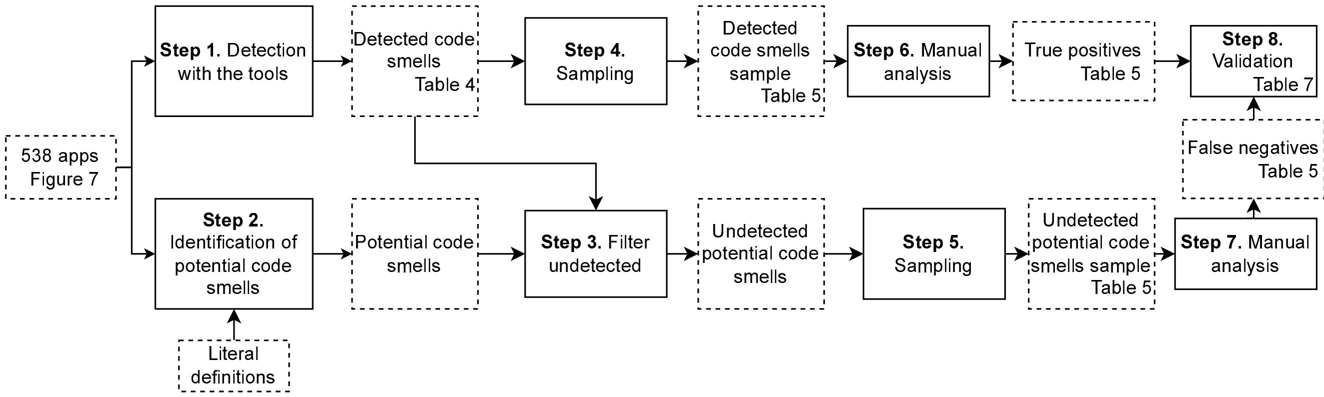
538 apps Figure 7
Step 1. Detection with the tools
Detected code smells Table 4
Step 4. Sampling
Detected code smells sample Table 5
Step 6. Manual analysis
True positives Table 5
Step 8. Validation Table 7
False negatives Table 5
Step 2. Identification of potential code smells
Literal definitions
Potential code smells
Step 3. Filter undetected
Undetected potential code smells
Step 5. Sampling
Undetected potential code smells sample Table 5
Step 7. Manual analysis

Fig. 8. Overview of the process.

we calculate the recall with the positive sample and negative sample. The recall is the proportion of true positives among true positives and false negatives $\frac{|TP|}{|TP \cup FN|}$.

6.5 Results

In this section, we will study and answer the research questions case by case.

6.5.1 RQ_1: Does DYNAMICS allows the detection of behavioural code smells ?

We investigate this research question through a quantitative study. We first present the number of detected code smells by the DYNAMICS tool, and compare them with those detected by PAPRIKA and ADOCTOR.

First, we examine the number of detected code smells by DYNAMICS depicted in Table 3. The number of detected code smells varies greatly depending on the type of code smell. We performed the detection for each input generator, as well as for the combination of the three input generators. Overall, we note that DROIDBOTX allows the detection of slightly more code smells than DROIDBOT, and DROIDBOT allows the detection of more code smells than MONKEYRUNNER. The combination of the three tools makes it possible to obtain more code smells, which corresponds to the union of the detected code smells by each input generator. The number of detected code smells depends on the number of events encountered, and therefore on the quality of the execution. The upper bound on detectable code smells is defined by the potential code smells. Thus, one cannot detect more code smells than there are classes containing encountered events, e.g. one cannot detect more DW code smells than there are classes containing encountered *acquire* calls. Factors such as the chosen input generator and the number of traces increase the number of detected code smells. The execution time also has an impact on the number of detected code smells. A longer execution time may allow more events to be reached, and therefore potentially more code smells to be detected.

This is however not always the case as shown in Table 3 that presents the number of detected code smells by DYNAMICS with 5-minutes and 10-minutes execution traces. Some code smells have indeed more instances when we double the execution time. But in other cases, the random nature of the input generator yields a reduced number of instances. In any case, combining traces from a 5-minutes execution with those from a 10-minutes execution further increases the total number of detected code smells, as presented in the last column "5min & 10min" of Table 3.

For the sake of comparison, we also examine the detected code smells by PAPRIKA and ADOCTOR presented in Table 4. N/A means that the tool does not detect the code smell. DYNAMICS returns fewer instances of code smells than ADOCTOR or PAPRIKA. For example, the difference is significant for DW where DYNAMICS detects 3 code smells compared to ADOCTOR, which detects 95 code smells. Also, for HAS DYNAMICS detects 15 code smells compared to PAPRIKA that detects 109 code smells. The difference is sometimes smaller, as for IOD where DYNAMICS detects 16 code smells compared to PAPRIKA that detects 18 code smells. As discussed in our previous empirical study [8], this may be due to the dynamic analysis allowing better precision and fewer false positives. But it may also be due to the fact that the coverage of events during the executions is not perfect and that some false negatives are present in the result of DYNAMICS.

> **RQ_1: DYNAMICS allows the specific detection of many types of behavioural code smells**. However, in terms of the number of instances, DYNAMICS detects fewer instances (as presented in Table 4) than ADOCTOR and PAPRIKA. As discussed in our previous empirical study [8], PAPRIKA and ADOCTOR reported false positives (about 30%). We now investigate the precision and recall of DYNAMICS to better understand these results.

6.5.2 RQ_2: Does DYNAMICS perform better in terms of precision and recall than the state-of-the-art tools ?

We investigate this research question through a quantitative study. We first present the precision and recall of DYNAMICS, and we compare them with those of ADOCTOR and PAPRIKA. Then, we discuss the recall of DYNAMICS by presenting the coverage of events during the executions.

Based on the manual analysis presented in the validation process depicted in Figure 8, we compute the precision and recall of DYNAMICS. Table 5 gives the precision and recall of DYNAMICS in comparison to ADOCTOR and PAPRIKA. Note

TABLE 3
Number of detected code smells with DYNAMICS depending of the input generator with an execution during five minutes and an execution during ten minutes.

Code Smell	DROIDBOTX		DROIDBOT		MONKEYRUNNER		DROIDBOTX DROIDBOT MONKEYRUNNER		
	5min	10min	5min	10min	5min	10min	5min	10min	5min & 10min
DW	3	3	3	1	1	3	3	3	3
HMU	221	266	235	268	185	227	294	319	324
HAS	6	2	5	2	9	2	14	5	15
HBR	2	3	4	2	1	2	5	5	8
HSS	2	1	0	2	2	1	3	2	4
IOD	10	11	5	8	9	7	14	13	16
NLMR	2004	2004	2004	2004	2004	2004	2004	2004	2004

TABLE 4
Number of detected code smells by DYNAMICS, PAPRIKA and ADOCTOR.

Code Smell	#Code Smells DYNAMICS	#Code Smells PAPRIKA	#Code Smells ADOCTOR
DW	3	N/A	95
HMU	324	573	N/A
HAS	15	109	N/A
HBR	8	305	N/A
HSS	4	63	N/A
IOD	16	18	N/A
NLMR	2004	1017	2999

that the precision and recall of DYNAMICS are computed based on the detected code smells using the traces obtained from the combination of the 5min and 10min executions (Last column of Table 3). Table 3 reports the results for all code smells except HBR and HSS. We choose to report only the results of the HAS code smell because the detection rules of HAS, HBR and HSS are very similar and differ only in the name of the method. Also, the number of detected code smells for HBR and HSS, in particular in PAPRIKA, is very high (see Table 4), which makes the manual analysis time-consuming. So, we specifically focus on HAS.

As presented in Table 5, the overall precision of DYNAMICS is high with an average of 92.8%. Also, DYNAMICS provides better precision than the other two tools for every behavioural code smell detected. Regarding the recall, DYNAMICS provides an average of 53.4% but the recall varies greatly depending on the code smells. Unlike the precision, the recall of DYNAMICS is not always better than the other two tools. This may be due to the fact that the coverage of events during the executions is not good. Indeed, some events associated with code smells may have not been encountered during the executions and thus, may cause the occurrence of false negatives, i.e. undetected code smells that should have been detected. Therefore, to verify this supposition, we study the coverage of events during the executions according to the input generators, the number of executions and the execution time.

To assess the reliability of our results, we posed the null hypotheses:

- $H_0(Precision)$: There is no statistically significant difference in precision between DYNAMICS and the state-of-the-art tools.
- $H_1(Recall)$: There is no statistically significant difference in recall between DYNAMICS and the state-of-the-art tools.

To test those hypotheses, we performed McNemar tests, comparing DYNAMICS results with those of PAPRIKA and ADOCTOR. The results of these statistical analyses are detailed in Table 5, which shows that, excepted the IOD code smell, all the obtained p-values are very low $< 0.00001\%$. Therefore, based on the results, there is enough evidence to reject the null hypotheses. We hence reject $H_0(Precision)$ because, apart from the case where the precision is 100% for each tool, the precision is much higher for DYNAMICS than for the other tools and we reject $H_1(Recall)$ because the recall was either way above or way below. Thanks to the McNemar test in Table 5, we know that this is not due to randomness, statistically speaking.

Table 6 shows the number of events (as specified in Table 1) encountered in the code (dataset containing 538 APKs) and in the execution traces generated by DROIDBOTX, DROIDBOT and MONKEYRUNNER. The events in the code are unique source code characteristics located statically in the app whereas the encountered events in the execution traces are unique events encountered during the executions as a result of dynamic analysis. The number of events varies slightly depending on the input generator used and greatly depending on the code smell. For example, for a 5 minutes execution, we encounter 272 HMU *Instantiation* events with MONKEYRUNNER while we encounter 308 *Instantiation* events for the HMU code smell with DROIDBOTX. Similarly, with DROIDBOTX we encounter 4 *Acquire* events for the DW code smell while we encounter 308 *Instantiation* events for the HMU code smell. The coverage of events is higher using DROIDBOT or DROIDBOTX than MONKEYRUNNER. However, it is possible to obtain better coverage of events with MONKEYRUNNER by increasing the number of actions per second as a parameter of the tool. In contrast, DROIDBOT and DROIDBOTX only allow one action per second. Also, combining the input generators, increasing the number of executions and/or increasing the execution time results in a slight improvement of the coverage of the events, just as it increases the number of detected code smells (see

TABLE 5
Precision and recall of DYNAMICS in comparison with ADOCTOR and PAPRIKA.

Code Smell	**#Detected CS / #Detected CS Sampled / #True Positives / Precision : $\frac{\|TP\|}{\|D\|}$**			**#Undetected Potential CS / #Undetected Potential CS Sampled / #False Negatives / Recall : $\frac{\|TP\|}{\|TP \cup FN\|}$**			**McNemar's test / χ^2 / p-value**	
	DYNAMICS	**PAPRIKA**	**ADOCTOR**	**DYNAMICS**	**PAPRIKA**	**ADOCTOR**	**Dyn vs Pap**	**Dyn vs aDo**
DW	3 / 3 / 3 / **100%**	N/A	95 / 46 / 9 / 19.6%	37 / 27 / 7 / 30%	N/A	8 / 8 / 5 / **64.3%**	N/A	24.143 / $< 0.00001\%$
HMU	324 / 69 / 44 / **63.8%**	573 / 30 / 10 / 33.3%	N/A	512 / 76 / 43 / **50.6%**	800 / 82 / 45 / 18.2%	N/A	254.429 / $< 0.00001\%$	N/A
HAS	15 / 11 / 11 / **100%**	109 / 46 / 16 / 34.8%	N/A	494 / 79 / 17 / 39.3%	424 / 77 / 10 / **61.5%**	N/A	50 / $< 0.00001\%$	N/A
IOD	16 / 16 / 16 / **100%**	18 / 17 / 17 / **100%**	N/A	114 / 41 / 12 / 57.1%	121 / 53 / 11 / **60.7%**	N/A	2.333 / $< 0.126633\%$	N/A
NLMR	2004 / 92 / 92 / **100%**	1017 / 87 / 87 / **100%**	2999 / 93 / 93 / **100%**	16 / 16 / 10 / **90.2%**	1370 / 90 / 89 / 49.4%	1074 / 88 / 85 / 52.2%	1363.003 / $< 0.00001\%$	1061.014 / $< 0.00001\%$
	Average precision of DYNAMICS : **92.8%**			Average recall of DYNAMICS : **53.4%**			1395.199 / $< 0.00001\%$	984.067 / $< 0.00001\%$

TABLE 6
Number of events encountered in the code and in the execution traces of the input generation tools on DYNAMICS during 5 minutes and 10 minutes.

Code Smell	**Event**	**In the code**	**DROIDBOTX**		**DROIDBOT**		**MONKEYRUNNER**		**DROIDBOTX DROIDBOT MONKEYRUNNER**		
			5min	**10min**	**5min**	**10min**	**5min**	**10min**	**5min**	**10min**	**5min & 10min**
DW	**Acquire**	45	4	4	3	2	4	4	5	4	5/45 (11%)
	Release	50	2	2	1	1	3	2	3	2	3/50 (6%)
HMU	**Instantiation**	1559	308	371	324	367	272	323	406	439	449/1559 (29%)
	Addition	2654	158	187	141	211	134	164	205	254	265/2654 (10%)
	Deletion	47	3	4	4	4	1	3	4	5	5/47 (11%)
	Clean	438	18	33	15	28	19	18	29	42	43/438 (10%)
HAS	**Begin/End**	703	90	113	89	103	68	91	128	132	143/703 (20%)
HBR	**Begin/End**	719	43	58	42	58	44	40	66	84	94/719 (13%)
HSS	**Begin/End**	134	25	31	31	37	27	28	40	43	48/134 (36%)
IOD	**Begin/End**	130	41	46	37	50	31	41	49	54	56/130 (43%)
	New	487	9	9	4	10	5	6	9	11	11/487 (2%)
NLMR	**Begin/End**	18	1	1	1	2	0	2	1	2	2/18 (11%)

Table 3). However, it can happen that the coverage does not improve. For example for the HBR code smell, with MONKEYRUNNER, we got 44 40 *Begin*/*End* events for the 5 minutes execution and 40 *Begin*/*End* events with 10 minutes execution. This is due to the fact that the actions generated by a given input generator are done randomly. As an example of the increase in the events coverage, with the combination of the three input generators with the 5 and 10 minutes execution, we reach 94 encountered events for the HBR *Begin*/*End* event, i.e., a significant increase of 62% (36 events) from the 58 events of DROIDBOTX, which was the highest. On the other hand, for IOD, we reach 56 encountered events for the *Begin*/*End* event, i.e., a slight increase of 12% (6 events) from the 50 events of DROIDBOT, which was the highest. Despite the fact that we have increased the execution time, the number of executions and the number of input generators, the coverage remains low. It only varies from 2% to at most 43% depending on the event. It seems important to explore other input generators more adapted in order to allow better coverage.

The study of the coverage of events confirms the supposition stated before that the recall is not that good because of the low coverage of events. Indeed, the existing code smells are bounded by the events in the code, e.g. 45 *Acquire* events for the DW code smell. On the other hand, the detectable code smells are bounded by the encountered events, e.g. 5 *Acquire* events for the DW code smell. There will be therefore undetected potential code smells that should have been detected, i.e. false negatives, as long the coverage is under 100%. For example, there are 40 *Acquire* events for the DW code smell that are not encountered by our tool that potentially relate to a code smell. As these false negatives are bounded by the events not encountered in the code, the lower the coverage, the more frequent the false negatives, and the lower the recall. However, one could expect the coverage of events to be directly related to the recall, but it is not as simple as that. While low coverage generally yields low recall, a lower coverage does not necessarily imply a lower recall. For instance, the HMU code smell, which has 29% coverage of the *Instantiation* event has a recall of 68.4% whereas the IOD code smell, which has 43% coverage of the *onDraw* methods has a recall of 57.1%. A detailed study of input generator approaches and their impact on recall falls outside the scope of this study but is surely worth further investigation.

RQ_2: **DYNAMICS allows the precise and accurate detection of behavioural code smells with a precision of 92.8% and a recall of 53.4%**. The recall depends strongly on the coverage of events during the executions. Although the precision is high, the recall is reasonable compared to the coverage of events. Increasing the number of executions and the time of the executions as well as combining the input generators will allow only a negligible to a slight improvement in the coverage. The coverage could be improved by using a dedicated input generator.

6.5.3 RQ_3: *Are there instances detected by the dynamic analysis that could not have been detected statically ?*

Through a qualitative study, we examine each behavioural code smell detected by DYNAMICS and discuss their dynamic nature to explain why they have been detected by DYNAMICS. We furthermore compare how the code smells are detected using static detection tools such as PAPRIKA and ADOCTOR as presented in Table 4 and Table 5, and highlight the importance of dynamic analysis. The results discussed are those obtained with the DROIDBOT, DROIDBOTX and MONKEYRUNNER traces simultaneously with an execution of 5 minutes and an execution of 10 minutes as presented in Table 3, Table 5 and Table 6.

DW: The DYNAMICS detection of the DW code smell provides a far better precision (100%) than ADOCTOR (19.6%) while the recall is worse (30% against 64.3%). The big difference in precision is explained by the fact that the DYNAMICS detection rule for DW explicitly verifies that a call to the *acquire* method is not followed by a call to the *release* method (whether it is within the same method or within a different method belonging to the same class or a different class) while ADOCTOR verifies only the presence of the *acquire* string. The low recall is explained by the low coverage of *Acquire* events during the execution ($5/45 = 11\%$ see Table 6).

More precisely, for ADOCTOR, the static detection rule mentioned in their reference paper [7] specifies that if a method using an instance of the class *WakeLock* acquires the lock without calling the release, a smell is identified. However, after inspecting the concrete implementation of the rule within the ADOCTOR tool, we found that the implementation rule uses a regular expression "$(.*)acquire(\\s*)()$" searching for the string "acquire" in the code, and not the specific *acquire* method of the *WakeLock* class. A large number of false positives with ADOCTOR refer to *acquire* methods that do not belong to the *WakeLock* class.

Within the execution traces, out of the 5 DW *Acquire* events encountered (see Table 6), three were related to a code smell and two were not. Regarding the two DW *Acquire* events not related to a code smell, we found that each *acquire* method is called and effectively followed by a *release* method. Each of the 3 *Acquire* events correctly detected as code smells by DYNAMICS is not followed by a call to a *release* method although the method is present in another method of the same class. This shows that the presence of the *acquire* and *release* methods are not sufficient to exclude them as code smells. A concrete call to these methods within the same *WakeLock* is required. Current static tools such as ADOCTOR are thus insufficient for the precise detection of this code smell.

HMU: The DYNAMICS detection of the HMU code smell provides a slightly better precision (63.8%) and recall (50.6%) than PAPRIKA (respectively 33.3% and 18.2%). These results are explained by the fact that the DYNAMICS detection rule for HMU explicitly identifies small *HashMaps* (less than 500 elements) and big *ArrayMap*/*SimpleArrayMap* (more than 500 elements) while PAPRIKA identifies only the presence of *HashMaps*. Also, whether for DYNAMICS or PAPRIKA, most of the *HashMaps* identified are small.

This preponderance of small *HashMaps* explains the slight difference between the precision and recall obtained by PAPRIKA that detects any use of *HashMap* as a code smell.

More precisely, within the execution traces, out of the 449 HMU *Instantiation* events encountered (see Table 6), only three are *ArrayMaps* and one is a *SimpleArrayMap*, all the other events are *HashMaps*. Among these *HashMaps*, most of them are small *HashMaps*. Only two are *HashMaps* that exceed 500 elements and five go beyond 100 elements. The three *ArrayMaps* and the *SimpleArrayMap* encountered are small and thus, do not relate to HMU instances.

From a more technical point of view, for the detection of HMU, we have identified two interesting technical aspects. The first one is that the executions generated by the input generators do not necessarily allow the structures to reach a sufficiently large size to identify the presence of an HMU. Indeed, an execution that leads to a sufficiently large size within a specific structure may often require doing specific actions in a loop, and input generators rather perform multiple random explorations. The second interesting technical aspect relates to the detection within the KOTLIN apps. Indeed, the detection of the HMU code smell gives at least a third more HMU instances when performed on KOTLIN apps. Indeed, the compilation of KOTLIN apps generates automatically for the *Activity* and *Fragment* classes a large number of methods containing *HashMaps*, not present in the source code. We excluded them from the sampling and consider other HMU instances. We plan to refine the detection of DYNAMICS in the next version to exclude the generated methods.

HAS: The DYNAMICS detection of the HAS code smell provides a far better precision (100%) than PAPRIKA (34.8%) while the recall is worse (39.5% against 61.5%). The big difference in precision is explained by the fact that the DYNAMICS detection rule for HAS considers the execution time of the method, while PAPRIKA uses the cyclomatic complexity and instruction count metrics. However, these last two metrics give a very crude estimation of execution time whereas DYNAMICS considers specifically the execution time of the method. The low recall is explained by the low coverage of *Begin/End* events during the execution ($143/703 \approx 20\%$ see Table 6).

Indeed, out of the 15 detected instances with DYNAMICS, 14 are not detected by PAPRIKA (Table 5). In contrast, out of the 143 HAS *Begin/End* events encountered during the execution (Table 6), 126 were not related to a code smell according to DYNAMICS. Also, among the 126 methods not related to a code smell according to DYNAMICS, 25 were detected as code smells by PAPRIKA. This shows that a short method with low cyclomatic complexity and a low number of instructions can take a long time to execute, and a long method with high cyclomatic complexity and a large number of instructions can execute quickly.

IOD: The DYNAMICS detection of the IOD code smell provides the same precision as PAPRIKA (100%) and the recall is almost even (57.1% for DYNAMICS against 60.7% for PAPRIKA). Although these results in terms of precision and recall are quite similar, the detected instances are not similar. DYNAMICS detects this code smell using one property divided into two parts. The first part of the property verifies if the execution time of the method is under $1/60$s. The second part of the property verifies if there is the initialisation of an object in the *onDraw* method during the execution. PAPRIKA identifies this code smell by analysing statically if there is an initialisation of an object in the *onDraw* method. Since the DYNAMICS detection rule uses two parts, we could expect to detect more code smells with DYNAMICS. However, both tools detect almost the same number of instances because DYNAMICS encounters fewer methods during the execution. Indeed, the recall of DYNAMICS is explained by the coverage of only 43% of the methods during the execution (see Table 6).

More precisely, among the 16 IOD code smells (see Table 5), 6 are detected due to the presence of a *New* event, 8 are detected because the execution time exceeds 1/60th of a second, and 2 were detected because of the presence of both the *New* event and the exceeding execution time. The 8 (6 + 2) instances detected by the presence of the *New* event should have been also detected by the PAPRIKA detection rule but it detects only 2 of them. This can be explained by the difference in the concrete implementations of the detection rule within PAPRIKA and DYNAMICS. The 8 instances detected by the exceeding execution time cannot be detected by static tools because of the dynamic nature of the related rule. This shows the benefit of the dynamic nature of our tool.

NLMR: The DYNAMICS detection of the NLMR code smell provides the same precision as PAPRIKA and ADOCTOR (100%) and the recall is better (90.2% for DYNAMICS against 49.4% for PAPRIKA and 52.2% for ADOCTOR). This difference in the recall is explained by the slight difference in the interpretation of the methods to consider. DYNAMICS detects this code smell using two rules, one static and the other one dynamic. The static rule verifies if an *onLowMemory* or *onTrimMemory* method is implemented within an *Activity* class. The dynamic property verifies if one of these methods releases less than 1024KB of memory. Firstly, DYNAMICS considers the *onTrimMemory* method in addition to the *onLowMemory* of any class that inherits directly or indirectly from *Activity* unlike ADOCTOR and PAPRIKA. Indeed, *onLowMemory* has been deprecated after the publication of these two tools. Secondly, ADOCTOR also considers the FRAGMENT class, which is another Android class that implements the *ComponentCallbacks* interface that defines the *onLowMemory* and *onTrimMemory* methods. Therefore, ADOCTOR detects more code smells than DYNAMICS. For its part, PAPRIKA excludes the *Activity* classes that inherit from another internal *Activity* of the app, which explains why it detects fewer code smells than DYNAMICS. All three approaches seem valid according to the interpretation of the NLMR code smell. However, the retrieval of potential code smells and thus the sampling is done according to our interpretation, which biases the recall.

More precisely, on the 2004 NLMR instances detected, 2002 NLMR code smells (true positives) are detected by the static rule and 2 (true positives) are detected by the dynamic property, meaning the encountered methods were not clearing caches properly. Indeed, only two methods are encountered during the execution, as shown in Table 6. We also notice that there are only 18 methods identified in the source code, so this code smell is likely to be rather present as each app has at least one *Activity*.

Hence, this code smell's detection does not allow us to evaluate the benefit of our dynamic approach since the vast majority of code smells are detected statically. Only two instances of code smells are detected due to the dynamic nature of DYNAMICS. The detection rules are therefore almost identical for the three tools, depending on the interpretation of the code smell.

To conclude, the qualitative study shows that DYNAMICS is indeed able to be applied to the three categories of behavioural code smells: (1) The behavioural code smells characterised by the misuse of a method call or a sequence of method calls during the execution such as DW; (2) The behavioural code smells characterised by runtime issues, such as a long execution time of a method or excessive use of the memory such as HAS, HBR, HSS, IOD or NLMR; (3) The behavioural code smells characterised by undesired data variations during execution, such as the size of a structure becoming excessively large, such as HMU.

RQ_3: DYNAMICS allows the **precise and accurate** detection of behavioural code smells **due to its dynamic nature**. For each behavioural code smell, DYNAMICS allows the detection of specific instances of code smells that were not possible to detect by static analysis. Such detection is possible for the code smells belonging to the three categories of behavioural code smells. DYNAMICS allows also the detection of instances of code smells that were possible to detect by static analysis.

6.6 Threats to Validity

Internal Validity. The main threat to internal validity could be the input generators tools used. Indeed, the results depend a lot on the execution traces obtained. We could have gotten more significant results, especially for HMU, if we had good event coverage with the input generators. Moreover, each input generator used is based on a degree of randomness, meaning it can generate different traces if we run them several times. Different traces can potentially lead to a different detection of code smells. However, we pre-selected the input generators tools from a set retrieved from the state of the art to select the ones that seemed to be the most efficient to deal with. Another threat to internal validity is that adding instrumentation-related instructions could slightly negatively impact performance and corrupt detected code smells. Those added instructions are minimal and a comparison of the execution time before and after instrumentation showed no significant difference.

External Validity. The main threat to external validity could be the dataset used for the validation. The use of good quality and mature open-source projects may cause some code smells to be almost absent. Most of these code smells can be found in apps under development. F-DROID provides us with a variety of apps, more or less mature and of various quality, which allows us to have a representative dataset.

Construct Validity. For some code smells, the thresholds applied to identify them may be subjective in nature, although the properties associated to the detection of the code smells are based on the definition and references from which the definitions are drawn. In addition, the interpretation of some code smells can also be subjective, such as the NLMR code smell where the property ensures that the *onLowMemory* method frees a certain amount of memory. Similarly, for the IOD code smell, the property ensures that the execution time of the *onDraw* method, must be executed in less than 1/60th of a second. However, the threshold values in DYNAMICS can be adjusted and adapted to the context of the app.

Repeatability/Reliability Validity. The results of the validation are repeatable and reliable because DYNAMICS and all necessary related tools are open-source and available in the artefacts [46]. The results are also available in our artefacts.

Implication. We proposed DYNAMICS, a tool-based approach to automatically detect behavioural code smells in Android by dynamic analysis. Both researchers and practitioners can benefit from our work because we provide (1) a formal specification for seven code smells, (2) a method and tool to detect behavioural code smells, which can be extended to any behavioural code smells and (3) an empirical validation on 538 real apps using manual validation. DYNAMICS can be used by researchers and practitioners to validate and improve the quality of mobile apps. The execution step, furthermore, does not need to necessarily be automated by input generators; an application can be run manually to detect behavioural code smells. User tests already integrated into the development of an application can also be used in our approach to detect behavioural code smells encountered during these tests. The use of input generators or predefined test scenarios can therefore enable practitioners to add DYNAMICS to continuous integration or continuous development. Each step has a different computational time. The specification step is only carried out once when defining code smells, so the time is insignificant. The instrumentation step time varies according to the size of the application, but is extremely fast, in seconds. The detection step evolves according to the size of the traces, but is also excessively fast, in seconds. However, the execution step can be quite long. In our experiment, this took up to 10 minutes. Longer or repeated executions may even be necessary to achieve better results, as shown by the event coverage in the validation. However, by using predefined scenarios, for example in user test scenarios already present in the test battery, this could be accelerated. A dynamic approach like DYNAMICS will therefore take longer than a static approach, especially if there is a need to integrate additional executions into a continuous integration.

7 Conclusion and Future Work

In this paper, we introduced DYNAMICS, a tooled-based method to detect behavioural code smells in mobile apps using dynamic analysis. DYNAMICS consists of four steps. First, we specify the behavioural code smells through events and LTL properties. Second, DYNAMICS takes as input a mobile app in the form of an APK to produce an instrumented app. Then, it executes the instrumented app to generate execution traces using input generator tools. Finally, it analyses the execution traces to detect behavioural code smells that manifest during the execution of the mobile app (by ensuring that the LTL properties related to behavioural code smells are preserved). We perform a validation on the detection of seven Android-specific code smells on 538 real mobile apps fetched from F-DROID. We applied a mixed-method analysis to understand the results of our validation. *Quantitatively*, we analysed the detection results of DYNAMICS on the 538 apps and compare them to ADOCTOR and PAPRIKA in terms of the number of instances, precision and recall. *Qualitatively*, we analysed in detail the code smells reported and compared them with the static analysis detection tools to highlight the importance of dynamic analysis. The validation indicates that DYNAMICS allows the precise and accurate detection of behavioural code smells due to its dynamic nature. DYNAMICS allows the detection of three categories of behavioural code smells. (1) The behavioural code smells characterised by the misuse of a method call or a sequence of method calls during the execution; (2) The behavioural code smells characterised by runtime issues, such as a long execution time of a method or excessive use of the memory; (3) The behavioural code smells characterised by undesired data variations during execution, such as the size of a structure becoming excessively large. Any type of code smells that fall into these categories can be detected by DYNAMICS. We believe that DYNAMICS is a tool that can be useful as a complement to traditional static tools to provide finer-grained detection of behavioural code smells. The lower recall with high precision of DYNAMICS combined with the high recall with lower precision of a static method suggests that an ideal solution would combine both in a clever way to outperform both dynamic and static detection. A study of the complementarity is one of the first conceivable future works. A study on the evaluation of DYNAMICS by gathering the opinions of developers is also envisaged, in order to determine what the possible improvement points are. This should be the subject of an empirical study, possibly controlled by monitoring the technical and human parameters. All the scripts and software required for replication to perform such an evaluation are available in the artefacts. As other future work, we intend to extend DYNAMICS to detect more behavioural code smells, especially by defining new behavioural code smells that benefit from our method. We also plan to work on dedicated input generators to improve the coverage of events, in order to reach a better recall for the detection of behavioural code smells. Finally, we plan to instrument apps that will be used by real users over a longer period of time. This would make it possible to have executions that are more realistic and significant and thus improving the event coverage and recall.

ACKNOWLEDGMENTS

We are grateful to C. Chekhaba, N. Khezemi and A. Sawadogo for their help on the manual validation. We especially thank N. Khezemi for her help on the selection of the input generators. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] "Number of apps available in leading app stores," https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 2022, online, Accessed: November 2022.
[2] "Number of mobile app downloads worldwide," https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/, 2022, online, Accessed: November 2022.
[3] M. Fowler, *Refactoring - Improving the Design of Existing Code*, 1st ed. Addison-Wesley, 1999.
[4] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design," in *ICSM*, 2005.
[5] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.
[6] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, "Detecting antipatterns in android apps," *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pp. 148–149, 2015.
[7] F. Palomba, D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia, "Lightweight detection of android-specific code smells: The adoctor project," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 487–491, 2017.
[8] D. Prestat, N. Moha, and R. Villemaire, "An empirical study of android behavioural code smells detection," *Empirical Software Engineering*, vol. 27, 2022.
[9] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, "Antipatterns: Refactoring software, architectures, and projects in crisis," 1998.
[10] G. Hecht, "Détection et analyse de l'impact des défauts de code dans les applications mobiles. (detection and analysis of impact of code smells in mobile applications)," Ph.D. dissertation, 2016.
[11] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," *Softwaretechnik-Trends*, vol. 34, 2014.
[12] "Arraymap," https://developer.android.com/reference/android/support/v4/util/ArrayMap.html, 2015, online, Accessed: November 2022.
[13] C. Haase, "Developing for android ii the rules: Memory," https://medium.com/google-developers/developing-for-android-ii-bb9a51f8c8b9, 2015, online, Accessed: November 2022.
[14] G. Mariotti, "Antipattern: freezing the ui with an asynctask," http://gmariotti.blogspot.com/2013/02/antipattern-freezing-ui-with-asynctask.html, 2013, online, Accessed: November 2022.
[15] ——, "Antipattern: freezing a ui with broadcast receiver," http://gmariotti.blogspot.ca/2013/02/antipattern-freezing-ui-with-broadcast.html, 2013, online, Accessed: November 2022.
[16] ——, "Antipattern: freezing the ui with a service and an intentservice," http://gmariotti.blogspot.com/2013/03/antipattern-freezing-ui-with-service.html, 2013, online, Accessed: November 2022.
[17] I. Ni-Lewis, "Avoiding allocations in ondraw() (100 days of google dev)," https://youtu.be/HAK5acHQ53E, 2015, online, Accessed: November 2022.
[18] "Onlowmemory," https://developer.android.com/reference/ android/content/ComponentCallbacks#onLowMemory(), 2020, online, Accessed: November 2022.
[19] M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 121–130, 2017.
[20] G. Rasool and A. Ali, "Recovering android bad smells from android applications," *Arabian Journal for Science and Engineering*, vol. 45, 02 2020.

[21] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, “[journal first] earmo: An energy-aware refactoring approach for mobile apps,” *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 59–59, 2018.

[22] M. Kessentini and A. Ouni, “Detecting android smells using multi-objective genetic programming,” *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 122–132, 2017.

[23] E. Iannone, F. Pecorelli, D. D. Nucci, F. Palomba, and A. Lucia, “Refactoring android-specific energy smells: A plugin for android studio,” *Proceedings of the 28th International Conference on Program Comprehension*, 2020.

[24] S. Habchi, R. Rouvoy, and N. Moha, “On the survival of android code smells in the wild,” *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 87–98, 2019.

[25] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe, “Andlantis: Large-scale android dynamic analysis,” *ArXiv*, vol. abs/1410.7751, 2014.

[26] M. Zheng, M. Sun, and J. C. S. Lui, “Droidtrace: A ptrace based android dynamic analysis system with forward execution capability,” *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 128–133, 2014.

[27] A. M. Fard and A. Mesbah, “Jsnose: Detecting javascript code smells,” *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 116–125, 2013.

[28] S. Kumar and J. Chhabra, “Two level dynamic approach for feature envy detection,” *2014 International Conference on Computer and Communication Technology (ICCCT)*, pp. 41–46, 2014.

[29] R. Vallée-Rai, P. Co, E. M. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot: a java bytecode optimization framework,” 2010.

[30] “Monkeyrunner,” https://developer.android.com/studio/ test/-monkeyrunner, 2017, online, Accessed: November 2022.

[31] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Droidbot: A lightweight ui-guided test input generator for android,” *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 23–26, 2017.

[32] H. N. Yasin, S. H. A. Hamid, and R. Yusof, “Droidbotx: Test case generation tool for android applications using q-learning,” *Symmetry*, vol. 13, p. 310, 2021.

[33] S. Hallé and R. Khoury, “Event stream processing with beepbeep 3,” in *RV-CuBES*, 2017.

[34] A. Pnueli, “The temporal logic of programs,” *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, 1977.

[35] A. Bartel, J. Klein, Y. L. Traon, and M. Martin, “Dexpler: converting android dalvik bytecode to jimple for static analysis with soot,” in *SOAP ’12*, 2012.

[36] “Dalvik bytecode.” https://source.android.com/devices/tech/dalvik/, online, Accessed: November 2022.

[37] “Jarsigner.” https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html, online, Accessed: November 2022.

[38] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: an input generation system for android apps,” in *ESEC/FSE 2013*, 2013.

[39] “Droidutan,” https://github.com/aleisalem/Droidutan, 2017, online, Accessed: November 2022.

[40] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Humanoid: A deep learning-based approach to automated black-box android app testing,” *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1070–1073, 2019.

[41] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.

[42] “Android debug bridge (adb).” https://developer.android.com/studio/ command-line/adb, online, Accessed: November 2022.

[43] “Logcat.” https://developer.android.com/studio/command-line/logcat, online, Accessed: November 2022.

[44] “Paprika project,” https://github.com/GeoffreyHecht/paprika, 2014, online, Accessed: November 2022.

[45] “adoctor project,” https://github.com/fpalomba/aDoctor, 2016, online, Accessed: November 2022.

[46] “Study artefacts.” https://doi.org/10.21227/49vr-wr08, 2022, online, Accessed: November 2022.